Proceedings of the Austrian Robotics Workshop 2025

An Adaptable Multi-Robot Support System for Disaster Response*

Laurent Frering[1] and Gerald Steinbauer-Wagner[1]

***Abstract*— In the recent years, many use-cases have been found for robots in disaster response operations, and many functionalities have been developed for those robots. But in order to facilitate the use of those robots in real operations, their usage have to be integrated at the mission level. In this work, we present our architecture for a multi-robot support system for disaster response operations. The proposed system's goal is to integrate agent-oriented programming for high-level decision-making with arbitrary robot platforms, refining goals into executable robot skills that are monitored and reasoned on. We focus on the software architecture and implementation details and provide details on the system capabilities and on the technologies used, and we outline the process for extending and adapting the proposed architecture to new projects. We discuss the different use-cases where the proposed system was deployed, and distribute its current open-source implementation: https://gitlab.tugraz.at/D214D39B6CEB7ECC/mrss**

***Index Terms*— Software Architecture, Multi-Robot System, Disaster Response**

I. INTRODUCTION

The use of heterogeneous robot teams for disaster response is gaining traction, with multiple recent experiments and deployments in different settings and with different robot types [1], including the use of Unmanned Aerial Vehicles (UAVs) and Unmanned Ground Vehicles (UGVs) in disaster response scenarios.

We previously proposed a generic architecture for a multi-robot support system aimed at providing first responders with a centralized situational picture obtained by a human-robot team comprising interactive goal-driven autonomous robots [2]. We deployed and tested this system during a field experiment simulating a firefighting operation in mountainous terrain, and gathered results and feedback from participating firefighters. In the chosen use case, a UAV equipped with color and thermal cameras was deployed in selected areas, highlighting detected hotspots. This was followed by sending a UGV equipped with a water tank to those hotspots, providing a water supply to firefighters in the field.

This field experiment was successful in tackling the use-case and rated positively by the firefighters, but also showed some limits in its autonomy and reliability. Those encouraging results lead us to continue upgrading the system and deploy it in two additional field tests, refining the software architecture for easier use and adaptability to new use-cases. The system matured into a software stack displaying different functionalities stemming from the requirements elaborated with end-users. Mainly, it integrates robust and proven technologies (such as MQTT for inter-process communication), provides a streamlined process to be adapted to new projects with different robots and communication protocols, and makes use of a Belief-Desire-Intention (BDI)-based reasoning scheme for goal-driven reasoning [3].

Building on this process, we propose here an updated version of this system architecture named MRSS (Multi-Robot Support System). The focus is on the software engineering, detailing the different modules and communication technologies. We show that MRSS is modular, and can be integrated with arbitrary robots and communication protocols. We also highlight its ability to integrate high-level goal-driven reasoning with actionable robot skills. We go over each component, detailing the design choices and implementation details. We also provide an open-source implementation of this system to help with future field robotics deployments.

Our goal is to propose a flexible architecture leveraging agent-based reasoning and multi-agent monitoring, able to be easily adapted to varied projects. This leads us to leave some implementation to the project developer, in particular regarding the communication with external components. To facilitate such adaptations, a streamlined process for extending the system to new projects is presented, focusing on isolating the necessary changes to specific parts and outlining the required tasks and their rationale. We thus aim at striking a balance between robustness and flexibility.

Finally, we detail different deployments of the proposed architecture and how it allowed end-users to manage complex robot systems in the field.

To summarize the contributions, we propose a software architecture for a multi-robot support system designed in the context of disaster response, facilitating the integration of autonomous robots with external components. We detail the different modules and provide an open-source implementation, and detail how they can be adapted to different use-cases by showcasing past deployments. As an additional byproduct, the reasoner component showcases how to integrate the Jason BDI platform [3] with MQTT to easily integrate with external components to generate percepts and realize blocking actions.

II. RELATED WORK

Over the last few years, many efforts have been made to deploy robot teams in disaster response scenarios. In addition

*This work was partially supported by the Austrian Research Promotion Agency (FFG) with the project KI-SecAssist.

[1]Laurent Frering and Gerald Steinbauer-Wagner are with the Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Graz, Austria. `laurent.frering, gerald.steinbauer-wagner@tugraz.at`

to the ones mentioned earlier, we refer the reader to our previous paper for an overview of those [2].

We focus here on projects and related work developing multi-robot system applicable to disaster response scenarios and providing different levels of reasoning.

The SHERPA project [4] had very similar interests, exploring different interaction modalities and control levels with heterogeneous robot teams. The main difference is their focus on teams composed of one human and multiple robots, with direct physical interaction and co-presence. While here the humans and robots may act close to each other, the focus is on the mission level, with centralized decision-making and the ability to directly interact with the robots if necessary.

The NIFTi project [5] focused on designing a user-centric system for multi-human multi-robot cooperation, with realistic field deployments. They iterated over the design over the course of the project, converging towards a robust architecture that proved successful in deployments. However, their system differs to ours by focusing on small robot teams and user-centric semi-autonomous robot skills, whereas we propose a more scalable system with less interaction. In addition to this difference in scope, there is a difference in specificity, as their system is a fully mature solution with tightly integrated components. Our system is less rigid (though less robustly designed for a given task), providing a platform for future developments and focusing on adaptability to different projects.

More recently, Copilot MIKE [6] is an assistant system for multi-robot operations, deployed in the DARPA Subterranean Challenge. It provides well-defined levels of autonomy for task automation, and makes use of a modular scheduler component. While similar to us, we differentiate between higher and lower level goal management, and provide modular interfaces to facilitate reusability.

More generally, the authors of [7] realize a survey of recent multi-agent human-robot interaction systems. They classify those systems in terms of team size, team composition, interaction model, communication modalities, and robot control. They highlight current challenges that are in line with our objectives, such as understanding better the factors influencing workload and situation awareness in multi-human multi-robot teams, the impact of having heterogeneous robots with varying levels of autonomy on human factors, and the importance of scalability and transparency.

III. PROPOSED ARCHITECTURE

MRSS contains four main components interacting with each other, including the robots or agents. The next paragraphs will describe the components' functions and intercommunication, with the full diagram available in Figure 1.

The World Model centralizes and processes the data from different sources, namely the User Interface for newly created objects, the Task Management System for robot status data, and the robots themselves for high-bandwidth data such as image streams and direct control commands. In general, the data is processed and stored for future use by the different components, with the World Model acting as a single source

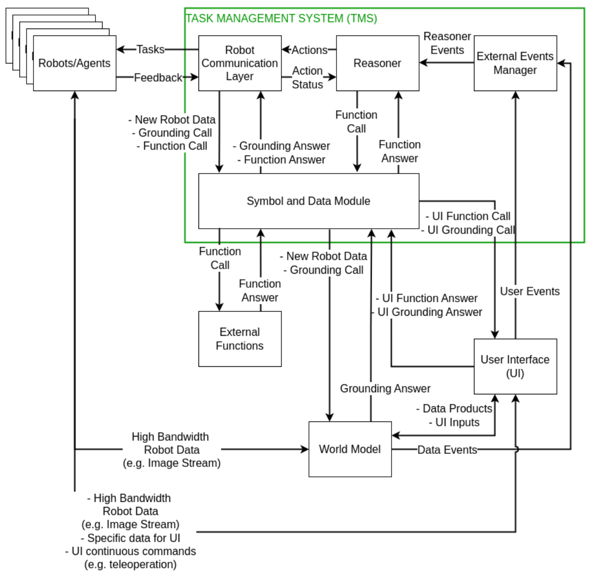


Fig. 1. Overview of the full system architecture. We focus on the Task Management System highlighted in green, and on how those components can be adapted to interact with different external modules.

of truth for the system. It is also equipped with rules dealing with current and new data, used to generate events for the Task Management System.

Those events are converted by the Reasoner submodule into actions for the robots to perform, with possible function calls to external modules during the processing. The actions refer to low-level goals, which are then assigned to one or more robots, and refined into actual robot tasks that the individual robots are equipped to perform. For example, the initial event could be triggered by a UAV detecting a hotspot; the Reasoner would refine this into an action for providing water close to the area, after checking that there is currently none in the vicinity. This action is then assigned to an available UGV, which receives the task of going to the corresponding area. In this way, the initial event is refined into more concrete operations at every level. In order to ground symbols into actual data, the World Model can be queried at any point in time. The Task Management System also monitors task execution and robot state, providing this information to the reasoner and World Model.

As shown in the figure, the User Interface is used to display data from the World Model and generate commands. In general, this information would go through the World Model before being forwarded to the Task Management System. This can also be bypassed in applications where the User Interface may be used to query non-critical information at runtime that is not planned to be used otherwise, for example to snap a picture of the current robot camera view. The same is true for the robot data. Information that is only to be used by the User Interface can be forwarded directly to it. In particular, those mechanisms are important for direct teleoperation, as it would often be the case that introducing the World Model in the middle of each operation would introduce undesirable latency. In order to safely switch to teleoperation, an event is initially generated so that the Task Management System can suspend all the actions currently assigned to the given robot. When it is notified about the end of the teleoperation, the robot is again considered available and the relevant actions are resumed.

Finally, the robots (or agents, as this architecture could easily accommodate non-embodied agents) are equipped with the ability to autonomously realize the tasks defined for this application, and provide feedback on their progress. They should also be able to update on their current state and communicate with the World Model or the User Interface directly for high bandwidth or low latency data.

In the proposed version, MRSS is able to consistently monitor a fleet of robots and react to changes in state or external events. The communication between the modules is thought to avoid conflictual commands, by centralizing the data in the World Model and minimizing the communication points between the different modules. It also allows for direct control of arbitrary robots, in an integrated way with autonomous decision-making.

IV. Implementation Details

We focus for the rest of the paper on detailing the Task Management System, responsible for the processing of external events and the allocation and monitoring of robot tasks. We will detail how it is implemented, and how it can easily be adapted to different robots, user interfaces, and world models by using modular adapters.

A. Architecture Setup and Communication

Every component of the Task Management System is running in a docker container, and they communicate with each other using MQTT (via an MQTT broker also running in a docker container). This allows for portability and isolation, and MQTT provides a proven and configurable framework that is reliable and easy to inspect and log. Apart from the Reasoner, the components are implemented in Python (version 3.1+). This choice was made to facilitate implementing project-specific modules, as Python3 provides many libraries implementing commonly-used communication protocols.

We will now detail the format of the messages exchanged internally between components. The External Events Manager receives the external events, which are defined as arbitrary messages with a mapping to first-order predicates using a project-specific adapter. Those first-order predicates represent the Reasoner Events, and are encoded as strings and sent over MQTT to the Reasoner. The Reasoner interprets those predicates as BDI percepts, which are refined into goals fed into the agent program. The outputs of the agent program are actions, which represent lower-level goals (either achievement or performative [8]) that the robot team can realize in the environment. They are also defined as first-order predicates and augmented with a Universally Unique Identifier (UUID) and a state variable representing the completion status. The actions are then sent to the Robot Communication Layer, which refines them into tasks, monitors their execution, and updates their status. The tasks are project-specific parameterized robot-specific skills to be executed, represented by the state machine defined in section IV-E. Finally, the Robot Communication Layer communicates with the Symbol and Data Module to forward it newly received robot data and send it requests to ground abstract objects IDs into python objects. Similarly, the Reasoner is also able to communicate with the Symbol and Data Module as an interface to call external functions. The Symbol and Data Module communicates with external functions and components using project-specific adapters.

The Docker Compose utility is used to easily start the whole stack. This way, every component is started at once and automatically configured for a given project.

B. External Events Manager

As discussed above, the External Events Manager is a Python-based component providing project-specific adapters for converting events with arbitrary representations and communication protocols to first-order predicates encoded as strings and sent over MQTT. It takes the shape of a python package, whose main functions are first to load an MQTT publisher to communicate with the other components, and second to load the project-specific adapter responsible for forwarding external events by using the MQTT publisher. This is done by making use of Python's dynamic import capabilities, to import the right module at runtime given the project name received from the Docker Compose configuration. The project-specific module implements an *External-Adapter* class that has access to the MQTT publisher, but otherwise has the responsibility to implement the custom logic to interpret the project events.

C. Symbol and Data Module

The Symbol and Data Module is similar to the External Events Manager, as it is mostly responsible with providing an adapter from external components to the Task Management System. It is also implemented as a Python package dynamically loading a project-specific module enabling bilateral communication to the internal (via MQTT) and external (via arbitrary communication protocols) components.

It manages requests over MQTT from the Reasoner and Robot Communication Layer, and also processes robot data received from the latter. On the other hand, it has to manage answers from the World Model and external functions over project-specific protocols.

The *DataLayer* class it implements thus provides MQTT callbacks triggered when receiving requests for external function calls, for grounding data via the World Model, and when receiving robot telemetry to be forwarded to the World Model. The exact list of callback functions and internal logic is flexible, as long as the module keeps track of the requests identifiers to send back the correct answer.

To enable effective symbol grounding, it is assumed that objects in the World Model expose relevant identifiers or key attributes to be used by external queries. Those can then be used in reasoning and for retrieving the linked data.

D. *Reasoner*

In order to provide high-level goal-driven reasoning, the Reasoner makes use of Agent-oriented programming. More specifically, we make use of the Belief-Desire-Intention (BDI) architecture that has a proven track record in providing complex goal-driven multi-agent systems. Fully detailing those concepts and their history is out of scope for the current paper, so we redirect the reader to the following review on those topics [9].

Following our previous work, the Jason platform BDI implementation [3] is chosen for its extensibility. Jason is implemented in Java, and the agents are programmed using the AgentSpeak(L) [10] agent programming language. An example AgentSpeak(L) program is shown on Listing 1, covering a simplified version of the program used in one of the use-cases that makes uses of simultaneous prioritized goals.

In order to integrate Jason into the architecture, we had to implement a few adaptations. First, we run it on a docker container as the other components. Second, we wrote a custom Jason environment that instantiates an MQTT client. This client is used to convert the events from the External Event Manager into BDI percepts (i.e. the ”inputs” of the agent program). Finally, we implemented a custom logic for the BDI actions: once selected by the agent program, an action is equipped with a UUID and forwarded over to the Robot Communication Layer using MQTT. An asynchronous latching mechanism is then used, so that the action is blocking until the latch is released. Another MQTT subscriber listens to feedback from the Robot Communication Layer, and releases the latch so that the action succeeds or fails according to the received feedback. Finally, a specific action named *rcl_goal_management* bypasses this mechanism, and is used to directly inform the Robot Communication Layer of meta action commands such as cancelling or suspending.

This way, we have a straightforward integration of the BDI agent program into the overall architecture, with actions naturally blocking until the underlying tasks either succeed or fail.

```
/* Initial beliefs */
isuav("r1").
available("r1").
/* Percepts */
+area_goal_received(G,P)[source(percept)] : true <- !
     coverarea("d1", G, P).
+goal_cancelled(G)[source(percept)] : currenttask(R,G,_) &
      isuav(R) <- -currenttask(R,G,_); .drop_intention(
     coverarea(R,G,_)); +available(R).
+prioritychange(G,P)[source(percept)] : .intend(coverarea(
     R,G,P)) <- .drop_intention(coverarea(R,G,P)); !!
     coverarea(R,G,P).
/* Plans */
+!coverarea(R, G, P) : isuav(R) & available(R) <- -
     available(R); +currenttask(R, G, P); cover(R, G); -
     currenttask(R,G,P); +available(R).
+!coverarea(R, G, P) : isuav(R) & not available(R) &
     currenttask(R, H, Q) & P>Q <- -currenttask(R,H,Q); +
     currenttask(R,G,P); .drop_intention(coverarea(R,H,Q))
     ; !!coverarea(R,H,Q); cover(R,G); -currenttask(R,G,P)
     ; +available(R).
+!coverarea(R, G, P) : isuav(R) & not available(R) <- .
     wait(2000); !!coverarea(R, G, P).
```

Listing 1. An example BDI Agent code for managing a single UAV with goal priority and cancellation. The Percepts are obtained via the External Event Manager, and respectively generate a *coverarea* goal, cancel an existing goal, or change the priority of an existing goal. The plans implement the behaviours for a *coverarea* goal: either sending the *cover* action to the Robot Communication Layer if the UAV is available, cancelling a previous lower-priority goal if necessary, or waiting and retrying if there is a current higher-priority goal.

E. *Robot Communication Layer*

The Robot Communication Layer is maybe the most complex component of the architecture. It receives new action commands from the Reasoner via MQTT, refines them into skills, and monitors their execution on the robots. It also communicates with the Symbol and Data Layer to forward robot data to the World Model, and to request the grounding of data or external function calls.

Once again, this component is implemented as a Python package, dynamically loading a project-specific module.

At the basic level, the Robot Communication Layer provides an abstract class to implement skills. Taking inspiration for existing skill models [11], skills are represented as a simple state machine progressing between the *Start*, *Run*, *Interrupt*, and *Finish* states. They can also acquire and release resources, though this mechanism was not yet tested in the deployments. A *SkillManager* class is available to interface with existing skills, running them in separate threads and managing their transitions. The individual skills and their behavior are left to the project programmer.

When an action is received, it is refined according to project-specific ”recipes”: a specific list of skill is instantiated, and data may be retrieved either from a previous execution (e.g. last waypoint reached for an area coverage skill), or by making a request to the Symbol and Data Layer. Each skill of the action is then started, and continuously monitored as part of the main loop of the module. The default behavior is akin to a logical AND: the action succeeds if all of its skills succeed, and fails if any of its skills fails. This can however be customized for each action by the developer. Once an action fails or succeeds, arbitrary data on its execution may be stored for future reference and the final action status is forwarded to the Reasoner over MQTT, so that the corresponding BDI agent plan can progress (or fail). Lastly, when the special *rcl_goal_management* action is received from the Reasoner, the corresponding action is directly transitioned to the corresponding state. This allows the Reasoner to bypass the normal behavior of the Robot Communication Layer if necessary in the high-level reasoning (for example, to directly interrupt an action or make it succeed).

V. Adaptation Process

We now detail and summarize the process for adapting MRSS to a new project or use-case. We go over all the parts that may be changed, highlighting the reasoning and specifications. A summary of this process can be seen in Table I.

To facilitate this process, a default project is implemented, providing a template for every part to be changed.

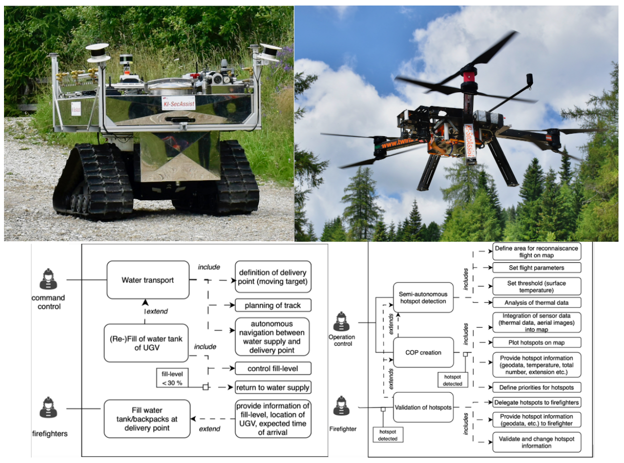
The first step is to create a new Docker Compose file for the project. The file should be named *docker-compose-{project_name}.yml* (with {*project_name*} to be replaced by the project name) in order to be conveniently started by the companion script which manages the clean starting, stopping, and rebuilding of containers. Using a separate Docker Compose file lets the components use different containers and images and allow for additional customization. The Docker Compose file can fully reuse the default one, but the user may modify it they need additional components to be started, or if they wish to use project-specific Dockerfiles instead of the default ones.

For the Event External Adapter, the user has to create a new module in the *external_adapters* folder named {*project_name*}*.py*. Similar to the default template, this module should implement an *ExternalAdapter* class, making use of the provided MQTT client to publish the Reasoner events. This class act as an adapter with the project-specific event representation and communication protocol.

Adapting the Reasoner consists only of adapting the Jason-related files. This includes the {*project_name*}*.mas2j* file, which should simply point to the agent file (and any addition that a knowledgeable Jason developer may use). The agent file, {*project_name*}*.asl*, is a standard Jason agent program. The only specific requirement is that, by default, it is assumed that the Reasoner performs action assignment to a specific robot. This means that actions are represented as first-order logic predicates in the shape *action(robot_id, goal_id)*. If the project requirements are different, this can be changed; however the Robot Communication Layer's action callback will need to be changed accordingly (see below).

For the Robot Communication Layer, the user has to create the project module *rcl_*{*project_name*}*.py* implementing the *RobotCommLayer* class in the *projects* folder. We suggest also creating a *rcl* {*project name*} *skills.py* module in the same folder to separate the main code from the skills definitions. To implement the skills, it is necessary to import and inherit from the *Skill* class from the *rcl_skill_model.py*. Each skill should have a custom implementation of the *start*, *run*, *finish*, and *terminate* functions. The *RobotCommLayer* class can be templated from the default one, but the user has to adapt the *refine_action* function with the project-specific action refinement recipes (i.e. which skills are started with a given action). Optionally, the user can adapt the actions' termination conditions in the *check_action_status* function, and has to adapt the *actionCB* callback function if the action representation in the Reasoner was changed.

Finally, the Data Layer simply requires a module in the *projects* folder implementing the *DataLayer* class. Similar to the default template, this class should implement MQTT callbacks for receiving requests and robot data from the Robot Communication Layer. Those callbacks should trigger the necessary communication to forward the requests to external components and populate the world model.



command control
Water transport
include
definition of delivery point (moving target)
planning of track
extend
(Re-)Fill of water tank of UGV
include
autonomous navigation between water supply and delivery point
control fill-level
fill-level < 30 %
return to water supply
firefighters
Fill water tank/backpacks at delivery point
extend
provide information of fill-level, location of UGV, expected time of arrival
Operation control
Semi-autonomous hotspot detection
COP creation
extends
extends
hotspot detected
Firefighter
Validation of hotspots
hotspot detected
includes
includes
includes
Define area for reconnaissance flight on map
Set flight parameters
Set threshold (surface temperature)
Analysis of thermal data
Integration of sensor data (thermal data, aerial images) into map
Plot hotspots on map
Provide hotspot information (geodata, temperature, total number, extension etc.)
Define priorities for hotspots
Delegate hotspots to firefighters
Provide hotspot information (geodata, etc.) to firefighter
Validate and change hotspot information

Fig. 2. Water transport UGV (top-left), hotspot recognition UAV (top-right), and use-case diagram for the first experiment, taken from [2].

VI. Deployments

Multiple iterations of this architecture were deployed over the last couple years in different projects. An initial prototype was used the first field tests of the KISecAssist project [2]. There, Austrian firefighters had access to a User Interface for managing one UAV and one UGV in a mountain wildfire scenario. They could define areas for the UAV to cover and map, where it may also detect hotspots. Those hotspots could then be used as targets for the UGV to navigate to. The UGV was equipped with a water tank, and firefighters could pump water out of it. Additionally, the UAV was able to interrupt its tasks to go back to the home base when its battery was low, and continue where it left off afterward. The UGV also automatically came back to the home base when the water level was low. All the goals could be sent with priorities, and the robots were able to suspend and resume goals accordingly. Pictures of the two robots and the use-case diagram for the experiment are shown on Figure 2.

This experiment used the technologies highlighted above, but this first version was not designed for customization and was therefore not containerized, and was implemented only for this use-case. The communication with the User Interface and the World Model was done via Rest API calls. The UGV was controlled via MQTT, and the UAV via MAVlink. Even though this first experiment highlighted different technical problems, it was still relatively successful and received an encouraging evaluation from the firefighters. It managed to showcase how it was possible to integrate all the components together with different communication technologies.

As a second step, MRSS was used as part of the AMADEE-24 Mars analog mission taking place in Armenia [12]. This experiment used a single mobile manipulator equipped with different sensors. For this application, the architecture was refined, making use of docker and providing insights on its adaptability by applying it to another use-case. There, the system communicated to a PostGIS database and to the robot via MQTT. The reasoner was responsible

TABLE I

SUMMARY OF THE ADAPTATION PROCESS TO NEW PROJECTS. PARTS IN ITALIC ARE OPTIONAL AND ARE RELATED TO DEEPER CHANGES TO THE SYSTEM. PROJECT_NAME IS TO BE REPLACED WITH THE PROJECT NAME.

Component	**Files to create**	**Specific parts to adapt**
Docker Compose	docker-compose-{project_name}.yml	*additional components* *custom Dockerfiles*
Event External Adapter	{project_name}.py	ExternalAdapter class
Reasoner	{project_name}.mas2j {project_name}.asl	*custom action definition*
Robot Communication Layer	rcl_{project_name}.py rcl_{project_name}_skills.py	skills inheriting from Skill class actions refinement process in refine_action function *actions termination conditions in check_action_status function* *action callback in actionCB function*
Data Layer	dl_{project_name}.py	MQTT request and data callbacks

for checking the legality of a given action according to the operational requirements. This second deployment of the proposed system was overall simpler in the number of technologies to integrate and in the high-level reasoning, but it highlighted how the system could be adapted in another use-case, with different requirements.

Finally, the last deployment was made for the final tests of the KISecAssist project. There, similar technologies were used as in the first deployment, but the system was fully updated according to the definitions above. The use-cases were also updated following firefighters' requests, notably including the requirement to directly teleoperate the UGV. This was straightforward to implement by opening a direct HTTP connection from the User Interface to the UGV. To maintain safety, the User Interface first notifies the Task Management System of the switch to direct teleoperation. This leads the Reasoner (and then the Robot Communication Layer via the *rcl_goal_management* action) to suspend all existing goals relating to the UGV. Then, a specific skill is used to switch the UGV to direct control mode. Once the teleoperation is done, the user would move to a safe spot and manually trigger the switch back to autonomous control on the User Interface. This lead the reasoner to resume all suspended goals so that the robot could proceed naturally.

Adding such a function highlighted the flexibility of MRSS. Indeed, by relying on the goal and concurrency management of the system, this behavior could be added with only a few lines of code in the expected modules and with very minimal debugging.

VII. CONCLUSION

We showcased and detailled a system for multi-robot control in practical deployments. The proposed architecture and implementation result from an iterative design, based on requirements by end users and considerations of software engineering principles. The main benefit of the system is its capacity to blend agent-oriented programming for high-level control with task-level robot control, in a flexible way that provides clear guidelines to adapt it to different projects. The proposed system was tested during field experiments in three different occasions, allowing for iterating over the design and adapt it to different use-cases. We plan to continue using and updating the system for new projects and to accommodate new capabilities such as resource management at the skill level. Moreover, the system would benefit for an integrated monitoring and debug tool, making use of the internal MQTT messages and providing test procedures.

REFERENCES

[1] J. Delmerico, S. Mintchev, A. Giusti, B. Gromov, K. Melo, T. Horvat, C. Cadena, M. Hutter, A. Ijspeert, D. Floreano, *et al.*, "The current state and future outlook of rescue robotics," *Journal of Field Robotics*, vol. 36, no. 7, pp. 1171–1191, 2019.

[2] L. Frering, A. Koefler, M. Huber, S. Pfister, R. Feischl, A. Almer, and G. Steinbauer-Wagner, "Multi-robot support system for fighting wildfires in challenging environments: System design and field test report," in *2023 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE, 2023, pp. 32–38.

[3] R. H. Bordini and J. F. Hübner, "Bdi agent programming in agentspeak using jason," in *International workshop on computational logic in multi-agent systems*. Springer, 2005, pp. 143–164.

[4] L. Marconi, C. Melchiorri, M. Beetz, D. Pangercic, R. Siegwart, S. Leutenegger, R. Carloni, S. Stramigioli, H. Bruyninckx, P. Doherty, *et al.*, "The sherpa project: Smart collaboration between humans and ground-aerial robots for improving rescuing activities in alpine environments," in *2012 IEEE international symposium on safety, security, and rescue robotics (SSRR)*. IEEE, 2012, pp. 1–4.

[5] G.-J. M. Kruijff, I. Kruijff-Korbayová, S. Keshavdas, B. Larochelle, M. Janíček, F. Colas, M. Liu, F. Pomerleau, R. Siegwart, M. A. Neerincx, *et al.*, "Designing, developing, and deploying systems to support human–robot teams in disaster response," *Advanced Robotics*, vol. 28, no. 23, pp. 1547–1570, 2014.

[6] M. Kaufmann, T. S. Vaquero, G. J. Correa, K. Otstr, M. F. Ginting, G. Beltrame, and A.-A. Agha-Mohammadi, "Copilot mike: An autonomous assistant for multi-robot operations in cave exploration," in *2021 IEEE Aerospace Conference (50100)*. IEEE, 2021, pp. 1–9.

[7] A. Dahiya, A. M. Aroyo, K. Dautenhahn, and S. L. Smith, "A survey of multi-agent human–robot interaction systems," *Robotics and Autonomous Systems*, vol. 161, p. 104335, 2023.

[8] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf, "Goal representation for bdi agent systems," in *International workshop on programming multi-agent systems*. Springer, 2004, pp. 44–65.

[9] R. Calegari, G. Ciatto, V. Mascardi, and A. Omicini, "Logic-based technologies for multi-agent systems: a systematic literature review," *Autonomous Agents and Multi-Agent Systems*, vol. 35, no. 1, p. 1, 2021.

[10] A. S. Rao, "Agentspeak (l): Bdi agents speak out in a logical computable language," in *European workshop on modelling autonomous agents in a multi-agent world*. Springer, 1996, pp. 42–55.

[11] C. Lesire, D. Doose, and C. Grand, "Formalization of robot skills with descriptive and operational models," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 7227–7232.

[12] S. Jusner, S. Moser, S. Schaffler-Glößl, R. Halatschek, M. Eder, and G. Steinbauer-Wagner, "A mission architecture for a human-robot collaborative planetary exploration cascade," in *2024 International Conference on Space Robotics (iSpaRo)*. IEEE, 2024, pp. 321–327.