MISSION PROGRAMMING FOR FLYING ENSEMBLES

COMBINING PLANNING WITH SELF-ORGANIZATION

Oliver Kosak

Dissertation
zur Erlangung des Doktorgrades
doctor rerum naturalium (Dr. rer. nat.)
der Fakultät für Angewandte Informatik
der Universität Augsburg, 2021

Erstgutachter:	Prof. Dr. Wolfgang Reif, Universität Augsburg
Zweitgutachter:	Prof. Dr. Bernhard Bauer, Universität Augsburg
Drittgutachter:	Prof. Dr.-Ing. Sven Tomforde, Universität Kiel

Tag der mündlichen Prüfung: 02.11.2021

Abstract

The application of autonomous mobile robots can improve many situations of our daily lives. Robots can enhance working conditions, provide innovative techniques for different research disciplines, and support rescue forces in an emergency. In particular, flying robots have already shown their potential in many use-cases when cooperating in ensembles. Exploiting this potential requires sophisticated measures for the goal-oriented, application-specific programming of flying ensembles and the coordinated execution of so defined programs. Because different goals require different robots providing different capabilities, several software approaches emerged recently that focus on specifically designed robots. These approaches often incorporate autonomous planning, scheduling, optimization, and reasoning attributable to classic artificial intelligence. This allows for the goal-oriented instruction of ensembles, but also leads to inefficiencies if ensembles grow large or face uncertainty in the environment. By leaving the detailed planning of executions to individuals and foregoing optimality and goal-orientation, the self-organization paradigm can compensate for these drawbacks by scalability and robustness.

In this thesis, we combine the advantageous properties of autonomous planning with that of self-organization in an approach to Mission Programming for Flying Ensembles. Furthermore, we overcome the current way of thinking about how mobile robots should be designed. Rather than assuming fixed-design robots, we assume that robots are modifiable in terms of their hardware at run-time. While using such robots enables their application in many different use cases, it also requires new software approaches for dealing with this flexible design. The contributions of this thesis thus are threefold. First, we provide a layered reference architecture for physically reconfigurable robot ensembles. Second, we provide a solution for programming missions for ensembles consisting of such robots in a goal-oriented fashion that provides measures for instructing individual robots or entire ensembles as desired in the specific use case. Third, we provide multiple self-organization mechanisms to deal with the system's flexible design while executing such missions. Combining different self-organization mechanisms ensures that ensembles satisfy the static requirements of missions. We provide additional self-organization mechanisms for coordinating the execution in ensembles ensuring they meet the dynamic requirements of a mission. Furthermore, we provide a solution for integrating goal-oriented swarm behavior into missions using a general pattern we have identified for trajectory-modification-based swarm behavior. Using that pattern, we can modify, quantify, and further process the emergent effect of varying swarm behavior in a mission by changing only the parameters of its implementation. We evaluate results theoretically and practically in different case studies by deploying our techniques to simulated and real hardware.

Kurzfassung

Der Einsatz von autonomen mobilen Robotern kann viele Abläufe unseres täglichen Lebens erleichtern. Ihr Einsatz kann Arbeitsbedingungen verbessern, als innovative Technik für verschiedene Forschungsdisziplinen dienen oder Rettungskräfte im Einsatz unterstützen. Insbesondere Flugroboter haben ihr Potenzial bereits in vielerlei Anwendungsfällen gezeigt, gerade wenn mehrere in Ensembles eingesetzt werden. Das Potenzial fliegender Ensembles zielgerichtet und anwendungsspezifisch auszuschöpfen erfordert ausgefeilte Programmiermethoden und Koordinierungsverfahren. Zu diesem Zweck sind zuletzt viele unterschiedliche und auf speziell entwickelte Roboter zugeschnittene Softwareansätze entstanden. Diese verwenden oft klassische Planungs-, Scheduling-, Optimierungs- und Reasoningverfahren. Während dies vor allem den zielgerichteten Einsatz von Ensembles ermöglicht, ist es jedoch auch oft ineffizient, wenn die Ensembles größer oder deren Einsatzumgebungen unsicher werden. Die genannten Nachteile können durch das Paradigma der Selbstorganisation kompensiert werden: Falls Anwendungen nicht zwangsläufig auf Optimalität und strikte Zielorientierung ausgelegt sind, kann so Skalierbarkeit und Robustheit im System erreicht werden.

In dieser Arbeit werden die vorteilhaften Eigenschaften klassischer Planungstechniken mit denen der Selbstorganisation in einem Ansatz zur Missionsprogrammierung für fliegende Ensembles kombiniert. In der dafür entwickelten Lösung wird von der aktuell etablierten Ansicht einer unveränderlichen Roboterkonstruktion abgewichen. Stattdessen wird die Hardwarezusammenstellung der Roboter als zur Laufzeit modifizierbar angesehen. Der Einsatz solcher Roboter erfordert neue Softwareansätze um mit genannter Flexibilität umgehen zu können. Die hier vorgestellten Beiträge zu diesem Thema lassen sich in drei Punkten zusammenfassen: Erstens wird eine Schichtenarchitektur als Referenz für physikalisch konfigurierbare Roboterensembles vorgestellt. Zweitens wird eine Lösung zur zielorientierten Missions-Programmierung für derartige Ensembles präsentiert, mit der sowohl einzelne Roboter als auch ganze Ensembles instruiert werden können. Drittens werden mehrere Selbstorganisationsmechanismen vorgestellt, die die autonome Ausführung so erstellter Missionen ermöglichen. Durch die Kombination verschiedener Selbstorganisationsmechanismen wird sichergestellt, dass Ensembles die missionsspezifischen Anforderungen erfüllen. Zusätzliche Selbstorganisationsmechanismen ermöglichen die koordinierte Ausführung der Missionen durch die Ensembles. Darüber hinaus bietet diese Lösung die Möglichkeit der Integration zielorientierten Schwarmverhaltens. Durch ein allgemeines algorithmisches Verfahren für auf Trajektorien-Modifikation basierendes Schwarmverhalten können allein durch die Änderung des Parametersatzes unterschiedliche emergente Effekte in einer Mission erzielt, quantifiziert und weiterverarbeitet werden. Zur theoretischen und praktischen Evaluierung der Ergebnisse dieser Arbeit wurden die vorgestellten Techniken in verschiedenen Fallstudien auf simulierter sowie realer Hardware zum Einsatz gebracht.

Acknowledgments

Writing this thesis would not have been possible without the great support, helpful discussions, and valuable ideas from the people surrounding me. At this point, I want to thank all of them for helping, guiding, and sometimes pushing me towards achieving this work.

First of all, I want to give special thanks to my advisor, Professor Dr. Wolfgang Reif, for providing me with the required environment for my studies. His guidance through many years of cooperative work allowed me to develop my own creative ideas for solving interesting problems and brought those thoughts back to focusing on the relevant aspects when needed.

I would like to thank all my colleagues and student assistants at the Institute of Software & Systems Engineering at the University of Augsburg, who provided the necessary productive environment for the extensive discussions that were necessary to ensure the quality of my work, never forgetting to also bring back the fun of the day-to-day work when thoughts got too caught up in the details. A special thanks go to my long-time office mate, Professor Dr. Alexander Schiendorfer, who was a never-ending source of knowledge for close to all scientific concerns I had to discuss. Thanks, Alex, for all the inspiring discussions on technical things and all other things of life we could talk about in an uncomplicated and often humorous way. Moreover, I want to explicitly thank my project, college Constantin Wanninger. Without your productive support in our project, I would have never been able to realize my ideas and get them grounded again when they got too overambitious. Thanks to Dr. Andreas Angerer, Dr. Alwin Hoffmann, and Dr. Hella Ponsar for their help in the starting time my work on this thesis. Your ideas and visions helped me finding my place in the research community. My thanks also go to Dr. Gerrit Anders and Dr. Florian Siefert for encouraging my interests in the area of research concerning self-organization.

I would also give special thanks to my family. Without the support of my parents Eva and Günter, I would not be where I stand now. You settled the base for all I achieved. Finally, my greatest thank goes to my wife Sarah, who supports me in every possible situation of our daily life and who encouraged me over and over again to not abandon the sometimes stony road to the goal. Last but not least, thank you, Lotta, for enriching my life with your happiness. You are my daily motivation.

Oliver Kosak

Contents

Chapter 1

Introduction

Summary. In this chapter, we motivate the necessity of our research on the problem of Mission Programming for Flying Ensembles and propose our approach of Combining Planning with Self-Organization for solving it. For achieving such, we focus on flying ensembles consisting of robots, i.e., Unmanned Aerial Vehicles, that can be physically reconfigured at run-time. To support our motivation, we give an overview of the current developments and related work in the different research fields of relevance, highlighting current issues in solving the problem. We further use the analysis of related work to extract important terminology and definitions from the literature that we further use in this thesis. Moreover, we depict the main challenges researchers currently need to face when dealing with the problem of Mission Programming for Flying Ensembles. Finally, we present the contributions of our approach of Combining Planning with Self-Organization for tackling these challenges and give the outline over the following chapters contained in this thesis.

Publication. Contents of this chapter have been published in Kosak [2015, 2017]; Kosak et al. [2018]; Kosak [2018].

1.1 Motivation

The goals and wishes humans want to achieve are uncountable. Unfortunately but also obviously, the abilities given to humans by nature for reaching those goals and wishes on their own are limited. This limitation, expressed to a specific and varying degree for each human, concerns sensing the natural environment in all its facets and acting in it and interacting with the objects present in that environment. On the one hand, these limitations are set because acting as desired can quickly become too strenuous or even impossible for a human, e.g., when trying to manipulate heavy objects, moving with a very high velocity, or trying to reach certain elevated or airborne places. On the other hand, the limitations concerning the sensing possibilities of humans cannot be defined that obviously. We currently know that the first categorization concerning human sensing abilities in the traditional senses of touch, hearing, sight, smell, and taste that was developed by the Greek philosophers Aristotle and Democritus (described in [Shields, 2012] and [Furley and Cole, 1970]) is outdated in modern physiology.

Today, we can already ascribe more sensing abilities to humans, e.g., thermoception or internal senses like the vestibular sense for body balance and acceleration [Nakamura, 2018], and there might be still other senses left to be found by researchers in the future. Nevertheless, we can still find sensing abilities available to other living creatures that humans clearly cannot make use of at all, e.g., the ultrasonic echolocation of bats [Holland et al., 2004] or the electroreceptor sensor pores of sharks [Heiligenberg, 2012]). Moreover, senses that are available to humans often are only expressed with a quality that is not sufficient for reaching the desired goal, e.g., for quantifying the concentration of some specific smell like dogs can perform with their much finer-grained senses [Willis et al., 2004]. Furthermore, there is a wide range of other sensing and acting possibilities that can be useful for humans and cannot be found in living creatures.

The insight that humans' abilities are limited, combined with creative ideas on how to use sensing and acting abilities other than one's own beneficially, serves as a permanent motivation for humanity to develop new technologies. To compensate for our deficiencies, intelligent systems like robots become our representatives in ever more situations. Specialized robots can help us reach our goals by providing us with some of the missing abilities desirable in certain situations.

Definition: Abilities and Capabilities While the term ability is typically ascribed to individuals and human skills and talents, the term of capability is more common in the context of organizations or technical systems [Gerkey and Matarić, 2004; Chevaleyre et al., 2006]. Thus, we speak of capability when we address robots or software agents.

Possible applications involving the support of robots cover a wide range of different use cases: The most commonly known applications are those from industrial manufacturing processes in which stationary robots are used for the manipulation of heavy objects, e.g., like they are necessary for automobile production [Smys and Ranganathan, 2019; Read, 2006; Hoffmann et al., 2009]. However, robots can also provide support and help in entirely different situations of our daily lives, e.g., for helping older adults with Alzheimer's disease [Wang et al., 2017b] or other medical issues [Beasley, 2012; Loh, 2018]. Moreover, with the steady progress in the miniaturization of electronics and improvements in control theory [Boubaker, 2013], the range of possible applications for mobile robots increased in the last decade. Especially the development of highly flexible, mobile, and flying robots opened up a completely new range of applications.

Definition: Unmanned Aerial Vehicle (UAV) In the following, we use the term Unmanned Aerial Vehicle (UAV) as a collective term for others synonymously used in literature, e.g., drone by Cacace et al. [2016]; Entrop and Vasenev [2017], quad copter/quadcopter by Berrahal et al. [2016]; Morgan et al. [2016], multi copter/multicopter by Vásárhelyi et al. [2014]; Brosy et al. [2017], or multi rotor/multirotor by Palomaki et al. [2017]; Autoquad [2018].

We understand an UAV to be a helicopter-like, flying vehicle that uses three or more rotors for a stable flight and that, in general, can independently move in every 3-dimensional direction, only restricted by their maximum velocity and inertia.

With their high speed, UAVs can reach distant and airborne places in a short time [SZ DJI Technology Co., 2021], provide camera-enabled bird-view perspectives [Hood et al., 2017] that can optionally be enriched with additional sensor information like infrared data [Entrop and Vasenev, 2017] and thereby help with searching and detecting objects of interest, among many other possible applications. The possibilities for using UAVs with different sensing and acting capabilities provided by specialized sensors and actuators are so diverse that providing an exhaustive list here would be out of scope, and we can only provide an excerpt of applications to illustrate their potential: By using appropriately equipped UAVs, we can collect data of interest in-situ, e.g., weather data [Wolf et al., 2017], and generate temperature and humidity profiles [Brosy et al., 2017]. Further, we can achieve a local overview of different gas concentrations like methane (CH_4) [Barchyn et al., 2017], carbon dioxide (CO_2) [Roldán et al., 2015; Andersen et al., 2018], nitrogen monoxide (NO) and dioxide (NO_2) [Villa et al., 2016b], or derive local particle matter (PM) measurements [Wang et al., 2020]. Besides using UAVs for such measuring purposes, their possibility for transporting objects is currently already making its way to productive systems, e.g., for post-delivery by the *Deutsche Post DHL Group* [DHL, 2019] or *Amazon.com Inc.* [Amazon, 2020], or planned to be enrolled in the future for distributing medical equipment [Momont, 2020; Müller, 2018].

In recent years, the potential of UAVs has been noticed by various research communities exploring the possibilities that can arise from *multiple UAVs* working together cooperatively in *ensembles*.

Definition: Ensemble While the term ensemble originates from art, describing a group of persons working together creatively and cooperatively focusing on a specific goal, e.g., interpreting music scores in a musical ensemble, forming choreographies in a dance ensemble, or acting roles in an ensemble cast, we use it to define a technical system aiming at a similar: working together in groups or teams of multiple entities in a cooperative manner following the same collective goal. In the context of robotics, we can find other well established terms describing similar forms of cooperation like Multi-Robot Systems (MRS) [Vig and Adams, 2006], Swarm-Robotic Systems [Şahin et al., 2008], or aggregates [Pianini et al., 2015].

Examples for the usage of ensembles that we can find in the literature are manifold, sometimes also involving Unmanned Ground Vehicle (UGV) or Unmanned Underwater Vehicle (UUV). Applications range from such intended for the Distributed Surveillance of an area of interest or objects of interest [Perez et al., 2013; Meng et al., 2015; Manyam et al., 2017; Gu et al., 2018], over such used for Environmental Monitoring [Dunbabin and Marques, 2012; Thenius et al., 2016; Duarte et al., 2016; Wolf et al., 2017; Tripolitsiotis et al., 2017], up to applications used for Search and Rescue (SAR) missions [De Cubber et al., 2013; Becker et al., 2013; Hussein et al., 2014; Cacace et al., 2016; Dominguez et al., 2017] and the related field of autonomous emergency response in Major Catastrophe Handling scenarios [Daniel et al., 2009; Scherer et al., 2015; Nedjati et al., 2016; Vallejo et al., 2020]. In each of these applications, users of the respective ensembles need to decide how to program the flying, driving, or swimming ensemble to achieve the individual user-defined goals in the respective application. Using ensembles in a goal-oriented manner means finding approaches to program ensembles, i.e., specify missions for them and approaches for executing those programs. Because we want to

focus on flying ensembles in this thesis, we define this as the problem of *Mission Programming for Flying Ensembles*:

Definition: Flying Ensembles A flying ensemble consists of more than one UAV. Concerning the specific mission the flying ensemble needs to handle, the number of UAVs in the ensemble can or must scale and increase accordingly. The UAVs that participate in the flying ensemble autonomously collaborate in a benevolent manner to achieve their common goals cooperatively.

Definition: Mission Programming We understand mission programming as the act of formalizing the application-specific common goal, i.e., *what* should be achieved in which situation (*when*), in a way that a respective flying ensemble can understand this formalization and execute the mission as intended. Therefore, we require the system we address the mission to can decide on which UAV or which group of UAVs should form the respective ensemble for executing the mission or a specific part of the mission, i.e., *who* executes the mission. Further, we require appropriate measures to execute the mission or part of the mission correctly (*how* to execute the mission) as intended by its programmer. In our opinion, mission programming thus is closely related to autonomous planning [Ghallab et al., 2004] and scheduling [Lawler et al., 1993] that aim at providing solutions to the question *who* should *when* execute *what*.

How such mission programming is performed and which solution is the best to be applied for each of the different sub-problems arising during solving it is very diverse in the broad set of applications. Further, different approaches propose individual solutions for specific sub-problems only, e.g., for instructing and commanding ensembles, for the coordinated execution of missions or specific parts of missions, supervising the execution of missions, or provide architectures for building such systems.

In this thesis, we analyze the problem of Programming Missions for Flying Ensembles, the sub-problems we mention above, and propose an integrated approach for solving it by *Combining Planning with Self-Organization*. Because definitions of Self-Organization (SO), the associated SO-Systems, and SO-Mechanisms are manifold and vary across research disciplines, we use the following definitions as the common foundation for discussions and descriptions in this thesis:

Definition: Self-Organization (SO), SO-Systems, SO-Mechanisms Di Marzo Serugendo et al. [2004] states that Self-Organization is characterized to work "without central control, and through contextual local interactions. Components achieve a simple task individually, but a complex collective behavior emerges from their mutual interactions." Babaoglu et al. [2007] further define that "Self-organizing systems work bottom-up. They are composed of a large number of components that interact according to simple and local rules. The system's global behavior emerges from these local interactions, and it is not easy to deduce the properties of the global system by studying only the local properties of

its parts. Such systems do not use internal representations of global properties or goals; biological or sociological phenomena often inspire them."A Self-Organization Mechanism describes one of those specific phenomena aiming to isolate local interactions responsible for it and rules relevant for reproducing it.

In the following text, we use the adjective self-organizing to describe a system or a part of a system where it is appropriate according to these definitions.

The solution we propose is inspired by the two main approaches we can find in literature, which we combine in a new fashion: We integrate the user-guided *top-down* specification, definition, and planning of missions with the ensemble-based and self-organized *bottom-up* execution of these missions. Therefore, we combine techniques originating from the field of traditional artificial intelligence like planning, scheduling, explicit control, and execution of missions [Russel and Norvig, 2014; Ed Durfree, 2013] with the paradigm of Self-Organization [Di Marzo Serugendo et al., 2004; Babaoglu et al., 2007] and its subarea of Swarm-Robotics Şahin et al. [2008].

While progress and achievements in both related research communities are inspiring, they also suffer from drawbacks making them less beneficial for the application in uncertain, real-world scenarios when used isolated. Tasks there are highly flexible and potentially require huge groups of agents to cooperate successfully for executing respective missions. In the following, we give an overview of related work from both mentioned disciplines for highlighting the potential and the drawbacks in Section 1.2. We find that approaches that combine results from both directions are currently sparse to find and often restricted in their possibilities. We then elaborate on the reasons and point out the challenges we need to face when combining approaches from both research perspectives in Section 1.3. In this thesis, we aim to close that gap by combining high-level programming of missions for flying robots using the autonomous planning technology with appropriate SO-Mechanisms we use for executing these missions. By doing so, we can provide the user of our system with the precise control and comprehensible execution of a mission where needed. At the same time, we can give control to the ensemble executing the mission by using SO-Mechanisms hiding the complexity of execution and exploiting beneficial properties like robustness and scalability where they are suitable. This way, we can profit from results achieved in both research disciplines and avoid most of the disadvantages of exclusively using one discipline. In Section 1.4, we enlist the contributions we deliver with our approach and give an overview of the rest of this thesis.

1.2 Related Work

We give an overview of the benefits and limitations of current approaches dealing with the problem of Mission Programming for Flying Ensembles to elaborate on the key challenges lying in it. An in-detail study and analysis of the related work relevant for our approach to solving the problem are then proposed in the respective chapters, each focusing on one specific aspect of our approach. In general, we can see two main research directions achieving steady progress in the field: Multi-Agent Systems (MAS)/Multi-Robot Systems (MRS) research and Self-Organizing Systems / Swarm-Robotic Systems research.

1.2.1 Programming Multi-Robot and Multi-Agent Systems

Researchers from the Multi-Robot Systems (MRS) and Multi-Agent Systems (MAS) community that also focus on UAV aim at different relevant aspects of the Mission Programming for Flying Ensembles. Because in literature, there exists some discrepancy in the usage of terms, we clarify our nomenclature by noting the following:

Definition: Agents and Robots The terms agent and robot are often used as synonyms [Shehory and Kraus, 1998; Gerkey and Matarić, 2004; Chevaleyre et al., 2006]. For the sake of clarity, in the following text, we speak of *agents* only instead of *agents and robots*. We make an exception from this only if there is the urgent need to differentiate between the *software agent* controlling the *hardware robot*. Thereby, we follow the taxonomy of [Di Marzo Serugendo et al., 2004]: "An agent is a physical (robot), or a virtual (software) entity situated in an environment that changes over time: the physical world or an operating system respectively."

Definition: Heterogeneity and Homogeneity Following literature [Gerkey and Matarić, 2004], we use the terms homogeneity and heterogeneity to define the similarity of the agent-controlled robots' hardware configurations. In that terminology, homogeneous robots consist of the same hardware configuration, i.e., the robot controlling agents provide the same capabilities, while heterogeneous robots consist of different hardware configurations, i.e., the robot controlling agents may provide different capabilities.

Much effort was put into the goal of generating powerful software architectures [Roldán et al., 2016; Malaschuk and Dyumin, 2020] and algorithms enabling the goal-oriented *top-down* instruction of the respective systems [Gancet et al., 2005; Lacroix et al., 2007; Roldán et al., 2015; Costelha and Lima, 2012; García et al., 2013]. One consensus derived from that research is that software architectures applied to Multi-Agent Systems (MAS)/MRS should be multi-layered when "task analysis, task negotiation, task execution, and task supervision becomes important" [Yan et al., 2013] (in this context, tasks can be seen as an equivalent to missions). With layered software architectures, we can profit from appropriate abstractions of functionality and complexity, e.g., abstract from concrete hardware specifications during high-level planning of missions [Yan et al., 2013; García et al., 2013]. Unfortunately, current software architectures often are designed use-case or hardware-specific, i.e., dedicated to one specific environment or robot. Here, we see the need for a software architecture enabling the flexible and non-use-case-specific application of an ensemble implementing it.

To tackle the difficulties of specifying missions, there exist some promising approaches that aim at simplifying instructing ensembles [Mottola et al., 2014; Koutsoubelias and Lalis, 2016; Dedousis and Kalogeraki, 2018; Lima et al., 2018]. Analyzing the approaches, we can identify essential properties required for specifying missions for ensembles. First, there is the need for appropriately defining the control flow of a mission-enabling sequential, parallel, conditional, and repeated executions, like they are proposed in parts by Mottola et al. [2014] and Lima et al. [2018]. Second, we require the possibility to address missions and parts of them to individual agents, groups of agents, all agents in the ensemble, or even swarms within the

ensemble [Lima et al., 2018; Dedousis and Kalogeraki, 2018]. Third, we need a mechanism to store and process the result of executions and reason autonomously on its influence on the mission's progress, i.e., to take care of the correct data flow [Mottola et al., 2014]. As our in-detail study and analysis of related work focusing approaches for programming missions for ensembles in Chapter 4 shows, there is no current approach integrating all the necessary properties. Further, our studies there show that there is not any approach that is flexible in the use-case it is applied to, which we are convinced is needed for tackling the problem of mission programming as we specify it.

Further, researchers focus the autonomous goal-driven plan generation for MAS/MRS [Brumitt and Stentz, 1998; Obst and Boedecker, 2006; Breitenmoser et al., 2010; Dominguez et al., 2017]. Often, approaches integrate the act of planning with that of the coordinated execution of generated plans [Myers, 1999; Gorniak and Davis, 2007; Sampedro et al., 2016; Ed Durfree, 2013]. The goal of this research is: *1)* to specify for an ensemble consisting of multiple agents, which actions should be performed in which order and under which environmental conditions and, i.e., manually describe the problem domain and autonomously generate good plans for it. Unfortunately, many top-down approaches for autonomous planning and the coordinated execution of respective plans in the research field of MAS/MRS become inefficient. Inefficient arises for that approaches when the amount of participating robots or agents increases [Erol et al., 1994], hardware configurations of agents differ from each other, i.e., become heterogeneous, [Gerkey and Matarić, 2004], or the agents are situated in a real-world setting instead of simulation or laboratory environments [Amigoni et al., 2005; Georgievski and Aiello, 2014]. Concerning planning and to avoid the inevitable planning-state explosion of state-space planners in real-world environments, state-of-the-art approaches use detailed expert knowledge, e.g., in the form of Hierarchical Task Networks (HTN) [Georgievski and Aiello, 2014]. These already have proven to be practicably applied in real-world applications [Georgievski and Aiello, 2015]. Thus, an approach for mission planning for flying ensembles can benefit thereof. *2)* for executing the generated plans, approaches are required to identify the specific agents that are capable of executing the plans (task-allocation), instruct those agents to execute the plans (task-assignment), and coordinate the execution of plans (task-coordination). Moreover, all mentioned functionality ideally should be performed in a non-centralized fashion to avoid single points of failures and bottle-necks in performance during the process [Gerkey and Matarić, 2004]. One good and practical solution for tackling the problem of task allocation is to perform a proper problem decomposition with market-based approaches using heuristics [Dias et al., 2006]. An approach for Mission Programming for Flying Ensembles should make use of that achievement.

Problems arise in MAS/MRS approaches when task execution is exposed to uncertainties or requires largely scaled ensembles. While there exist many solutions providing sophisticated failure detection and troubleshooting mechanisms to handle uncertainties, e.g., failure of agents [Brooks et al., 2016], integrating these in an approach for Mission Programming for Flying Ensembles increases complexity and computational overhead [Hudziak et al., 2015]. Also, larger scaled applications (e.g., scaled spatially) heavily increase complexity in coordinating ensembles that are scaled [Steghöfer et al., 2013; Kosak et al., 2015] similarly. The effort put into large-scale applications, e.g., the Intel Drone Shows [Intel, 2021], is immense, and coordination can only be performed by centralized, high-performance computing solutions that calculate all UAVs' trajectories beforehand. Here, approaches originating from the field of traditional artificial intelligence reach their limit when coordination of missions needs to be performed within the ensemble, i.e., on-board of one or many UAVs in the ensemble. This is

where approaches from the SO and Swarm-Robotics community show their strength.

1.2.2 Programming Self-Organizing and Swarm-Robotic Systems

Researchers from the community of SO and Swarm-Robotics Systems investigate new technologies for commanding and coordinating groups of UAV, looking at the same problems as the MAS/MRS community, but from a different perspective [Tahir et al., 2019; Campion et al., 2019; Bürkle et al., 2011]. Following the definition of SO in Section 1.1, researchers focusing on software architectures and algorithms for SO or Swarm-Robotics Systems emphasize the local interactions of the individual agents with their respective neighbors [Barca and Sekercioglu, 2013; Şahin et al., 2008]. Because the term *swarm* is not used uniformly in the literature and thus not always as we intend it (e.g., Dedousis and Kalogeraki [2018], Sampedro et al. [2016] and Koutsoubelias and Lalis [2016] already use the term swarm when they speak of a group of agents instead of individuals only), we clearly state our understanding of it here.

We condense the following definitions from the systematic reviews performed by Şahin et al. [2008], Barca and Sekercioglu [2013], Brambilla et al. [2013], and Nedjah and Junior [2019]:

Swarm Behavior Swarm behavior can be found in nature. It describes the emergent effect that an observer can see on the macro-scope produced by the interaction of the individuals in the swarm on the micro-scope. Individuals in the swarm execute local behavior, often considering their direct neighborhood only. This allows swarms to scale in size, e.g., for increasing their spatial distribution while still achieving the macro-scope goal immensely. Further, there is no need for a central instance for controlling executions within the swarm to achieve its emergent effect. Because swarms lack such central instance, they are robust to failures of individuals in the swarm.

Swarm Algorithm A swarm algorithm is the adaptation of a swarm behavior for a technical system. Like individuals in a swarm found in nature, agents in a technical swarm execute local rules to produce an emergent effect cooperatively [Şahin et al., 2008]. If this emergent effect can be measured and further post-processed, we speak of goal-oriented swarm behavior.

Swarm Robotics Swarm robotics, thus, is applying swarm algorithms to robots aiming at transferring the beneficial properties of swarm behavior to controlling groups of cooperative robots. Following Brambilla et al. [2013], "Swarm robotics is an approach to collective robotics that takes inspiration from the self-organized behaviors of social animals. Through simple rules and local interactions, swarm robotics aims at designing robust, scalable, and flexible collective behaviors for the coordination of large numbers of robots."According to the survey on Swarm-Robotics performed by Barca and Sekercioglu [2013], a swarm "refers to a large group of locally interacting individuals with common goals. It is used to describe all types of collective behaviors even though it brings up associations to joint movement in space."

By adopting natural phenomena for technical systems, researchers try to beneficially make use of properties that are inherently available in many natural systems that emerge *bottom-up*, e.g., scalability, robustness, and the ability for making local decisions without central control [Di Marzo Serugendo et al., 2004; Babaoglu et al., 2007]. One of the most famous examples for such successful adaptations are those of adapting the local rules in ant colonies for robustly solving classical problems of computer sciences like the traveling salesman problem [Bianchi et al., 2002] or the shortest path problem [Dorigo et al., 2006], even if the problem size is scaled up. For solving different instances of the search problem, there exists a multitude of applications of the Particle Swarm Optimization (PSO) algorithm applied to different use cases [Zhang et al., 2015]. Some of them even include UAVs used in real-world [Sánchez-García et al., 2019; Na and Yoo, 2019] and simulation [Skrzypecki et al., 2020; Lee et al., 2018]. Other examples show that we can further adapt the self-coordination abilities of swarms of birds or fish to technical systems for reducing the coordination efforts that are useful when commanding huge groups of UAVs [Sánchez-García et al., 2019; Na and Yoo, 2019; Vásárhelyi et al., 2014]. There also exist approaches that aim at generating software controllers for individual swarm agents with genetic algorithms and other learning techniques, e.g., for coordinated motion in general [Trianni, 2008], foraging tasks [Pérez et al., 2017], searching and acting tasks [Dorigo et al., 2013]. Moreover, there are approaches enabling swarm agents to reach their goals with in-detail engineered situation-specific solutions, e.g., achieving the shaping of different geometric forms on the floor with problem-specifically designed robots (Kilobots) [Rubenstein et al., 2014].

The analysis of approaches from the SO/ Swarm-Robotics community shows that most of them unite the properties of providing robust and scalable solutions for the specific problem they tackle. In our opinion, an approach for Mission Programming for Flying Ensembles should exploit these beneficial properties to tackle the issues affecting exclusive MAS/MRS approaches we mentioned above. However, the solutions found by SO/ Swarm-Robotics community also have their drawbacks. Our studies on the related literature show that it is challenging to transfer the solutions found for specific problems to other applications than they were designed for. This is a drawback because slightly different defined tasks require the time-intense adaptation of existing approaches, including high engineering effort [Hamann et al., 2016]. Thus, there is an urgent need for a more general solution to program swarm behavior. Only a few approaches address this problem by proposing programming languages for swarms [Pinciroli and Beltrame, 2016], ensembles [Ashley-Rollman et al., 2007], and aggregates [Pianini et al., 2015]. While they reduce the required effort to develop and apply new swarm behavior, they share the identical drawback specifically designed approaches suffer from: After accomplishing a single swarm behavior, agents in the swarm are often not able to post-process the results of the swarm behavior's execution [Barca and Sekercioglu, 2013]. For achieving more complex goals using swarm behavior, we thus require the integration into a broader task-orchestration framework. While this need was already mentioned by Dedousis and Kalogeraki [2018], to the best of our knowledge, it was not implemented in any existing approach up to now. To close that gap, we see the urgent necessity to integrate results from the SO/ Swarm-Robotic Systems community into an approach for Mission Programming for Flying Ensembles.

1.3 Research Challenges

After studying the current state of research, we can now summarize the key challenges for Mission Programming for Flying Ensembles.

1.3.1 Increasing the Flexibility for Different Applications and Domains

From an observing perspective, many related approaches currently address similar problems in different granularity. New agents with new capabilities have to be designed, built, and programmed with individualized software for each new application. This leads to a very diverse field of approaches (cf. Figure 1.1) caused by different environments (underwater, ground, air, space), application scenarios (e.g., Environmental Monitoring, SAR, Distributed Surveillance, or, Major Catastrophe Handling), agent configurations (homogeneous or heterogeneous), and coordination paradigms applied for missions (approaches focusing on MAS/MRS, SO-Systems/Swarm-Robotic Systems, or manual). Consequently, we see the number of needed agents and robots growing proportionally with the number of different missions and use cases. In the context of Swarm-Robotic Systems, the systematic analysis of Nedjah and Junior [2019] comes to a similar conclusion and states it as the main drawback causing that "the field of swarm robotics is not yet mature enough, mainly due the absence of a universal methodology and generic robots that can be used in any, or at least in many, applications." To cope with this issue, an approach for Mission Programming for Flying Ensembles needs to address the challenge of achieving flexibility for different applications. It needs to apply to different types of environments, application scenarios, agent configurations, and coordination paradigms so those in-depth adjustments are not necessary when conditions for its use change. This requirement holds for the overall approach, i.e., concerning mission specification, mission planning, mission assignment, mission execution, and especially for the software architecture supporting all these mechanisms.

1.3.2 Distributed Deployment to Flying Ensembles

Flying ensembles are often meant to be used in outdoor environments. Furthermore, in many scenarios where flying ensembles should be used, there might not be a continuously stable communication connection between the agents in the ensemble and a ground control station for controlling the agents in a centralized manner. Especially in emergency cases, e.g., Major Catastrophe Handling scenarios like that treated by Daniel et al. [2010], and Sánchez-García et al. [2019], there even might not be an external communication channel with cellular connections. This increases the need for decentralization concerning the mechanisms integrated into an approach for Mission Programming for Flying Ensembles. The challenge here is to deploy as much computational intelligence on the device, i.e., directly on the UAVs, as possible. Possibilities here are limited when working with flying ensembles. Because agents in flying ensembles typically are restricted in the load they can transport, this also impacts the maximum computational power that each agent in the ensemble can provide. Typically, instead of high-performance processing units, only small-sized single board computers can be transported by UAVs [Li and Ling, 2015; Landolsi et al., 2018]. This is also due to the goal of having as much additional load capacity as possible for carrying mission-specific equipment, i.e., sensors and actuators required to enable agents to execute specific capabilities. Because the computational power of such single board computers is limited to a certain degree (the currently very popular

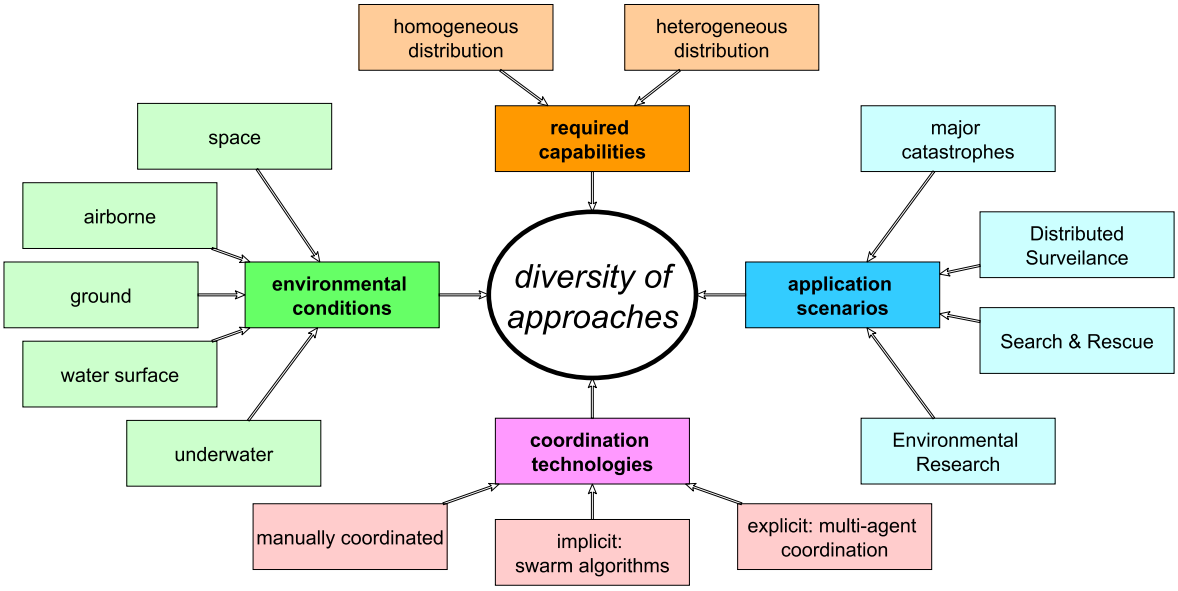


Figure 1.1: An excerpt of reasons causing the diversity of approaches found in literature concerning research done on technologies dedicated to autonomous ensembles.

Raspberry Pi, e.g., is powered by a 1.5 GHz quad-core processor [Raspberry, 2021]), algorithms used in an approach for Mission Programming for Flying Ensembles need to be light-weight. Instead of centralized complex computation for the whole ensemble or parts of it, an approach for Mission Programming for Flying Ensembles must be distributed to the ensemble's agents by proper and goal-oriented problem decomposition and division-of-labor approaches.

1.3.3 Appropriate Methodologies for Designing Missions

Of course, an approach for Mission Programming for Flying Ensembles must provide sufficient possibilities to instruct the ensemble as intended by the programmer. To enable the programmer to specify the mission/program flow in the way it is needed in a specific scenario, there is the need for maximum flexibility in mission design. Providing such flexibility, appropriate instructions, statements, and function calls for specifying missions addressing ensembles is required. These need to be powerful enough to enable the programmer to

- define sequential, parallel, repeated, and conditional mission flow situation-specific [Gutmann and Rinner, 2021]. This means, according to the situation of a flying ensemble that finds itself situated in a dynamic environment, the programmer should be able to specify which actions are required to be executed by the ensemble in which order.
- address missions and parts of missions to individual agents, groups of agents, or even swarms within the ensemble. This helps to avoid unnecessary programming overhead, e.g., in case a programmer wants all agents from an ensemble to execute the same task or to exploit scalability and robustness delivered by swarm behavior, there need to be reasonable possibilities to program such without the need for individual instructing every single agent [Lima et al., 2018]. This also requires that the programmer can first define the required agent groups and swarms within the ensemble.

- store and process the result of instructions and functions executed by individual agents or agent groups within the mission. This means that results derived by the ensemble during execution in an earlier part need to be available for evaluating decisions or for using them as parameters of other instructions in a later part of the mission.

1.3.4 Handling Complexity in Mission Planning

Further, an approach for Mission Programming for Flying Ensembles also needs to provide appropriate mechanisms for enabling an ensemble to decide which part of the designed mission needs to be executed in a specific situation. This is necessary because not all parts of that mission might be relevant in every situation. If e.g., a mission specifies that a flying ensemble should collectively measure some environmental parameter of interest at a given location, the ensemble first needs to move to that location. Depending on its current situation, i.e., whether the ensemble is already correctly positioned when the mission is activated or not, the ensemble does not need to execute this moving operation. To enable such decision making, we know that automated planning can be applied [Ghallab et al., 2004; Brenner and Nebel, 2009; Ed Durfree, 2013]. To enable this, an approach for Mission Programming for Flying Ensembles needs to provide the possibility to generate situation awareness when necessary. This, combined with the user-defined mission, is required as input to the planning problem. Automated planning unfortunately only comes by high cost: For example, creating plans that define the procedure of execution for missions is NP-complete [Erol et al., 1994] while allocating plans to agents is NP-hard, worsened through "larger team sizes and greater heterogeneity of robots and tasks" [Gerkey and Matarić, 2004]. Dealing with the challenge described in Section 1.3.1 arises the need also to handle both of these factors of increasing computational complexity. On the one hand, we need to handle "larger team sizes" if large spatial applications require such, e.g., when applying the control paradigm of swarm behavior, and, on the other hand, situations where "heterogeneity of robots" in an ensemble can occur. This is because different capabilities can often only be provided by different sensors and actuators, which soon may generate load capacity problems in flying ensembles (overloaded agents might not be able to fly anymore).

1.3.5 Hiding Complexity in Mission Execution

When a plan for an ensemble in a given situation is finally derived from a mission with automated planning, the next urgent challenge is *1)* to correctly assign that plan to agents that are capable of working on it and *2)* to further coordinate that execution, if necessary. While the underlying problem of task allocation for autonomous agents and MAS has already been in the focus of many researchers in the past [Shehory and Kraus, 1998; Mosteo and Montano, 2010; Hussein et al., 2014; Vig and Adams, 2006], the challenge described in Section 1.3.1 requires to adapt solutions found earlier to also work for agents that are flexible concerning the set of capabilities they have available during a mission. Given a situation where agents are not sufficiently equipped for working on a mission, there is currently no approach in literature for dealing with it. Because we suppose such situations emerge very often when dealing with the challenge defined in Section 1.3.1, there is the need for appropriate countermeasures enabling ensembles to progress work also under circumstances as described above. As this further complicates finding plan-compatible ensembles and plan execution, we require solutions that automate most of the process and only involve the system's user when this cannot be avoided or explicitly desired by the user.

1.3.6 Using Swarm Behavior to Handle Scale and Uncertainty

Caused by the challenge we describe in Section 1.3.1, an approach for Mission Programming for Flying Ensembles must also be applicable in spatially large or uncertain environments. While not all parts of a mission might be subject to this challenge, the approach must deliver scalability and robustness where they are urgently required. Swarm behavior is known to provide such properties when appropriately applied to technical systems in swarm algorithms and robots. While swarm behavior has already been successfully applied to technical systems in the past [Barca and Sekercioglu, 2013; Brambilla et al., 2013; Nedjah and Junior, 2019], the actual challenge is to achieve a goal-oriented embedding of swarm algorithms into our approach. Thus, we need to face the currently prominent high specialization of current swarm algorithms dedicated to specific tasks only. As we already highlighted in Section 1.2, most current approaches follow the principle of designing *one specialized swarm algorithm for each swarm behavior*. This strategy shows up one big drawback: Every time new swarm behavior is found in nature promising to be beneficial within a technical system causes a high engineering effort to design a new swarm algorithm. To reduce this overhead, we need to face the challenge of identifying and realizing a more general approach. We require to engineer proper swarm behavior by developing swarm algorithms for technical systems in general and flying ensembles in special. In this context, to be useful means for a swarm algorithm not only to reproduce the behavior identified in nature but also to be able to *1)* measure the result of a swarm algorithm's execution if this algorithm self-stabilizes or *2)* to continuously exploit the swarm algorithm's execution to derive goal-oriented intermediate results. Thus, we need to identify a general pattern for expressing goal-oriented swarm behavior algorithmically, which we can use flexibly in different contexts.

1.3.7 Applying Swarm Behavior to Heterogeneous Ensembles

Addressing the challenge we describe in Section 1.3.6 with swarm behavior combined with the challenge we describe in Section 1.3.1 arises another challenge to take. Swarm behavior in nature and its adaptations for technical systems is typically achieved by homogeneously configured agents. If different behavior is required in a swarm because of different goals in different tasks, other agents with different capabilities are required to execute these tasks. In some ant colonies where a variety of different tasks need to be performed, ants with different morphological appearances and abilities are needed for specific tasks like foraging, fighting against intruders, brood care, or reproduction, each forming a subsystem of homogeneous entities best suited for their respective collective task [Hartmann and Heinze, 2003]. Other environments even show up completely different species of ants having other morphological appearances for the same task [Véle and Modlinger, 2019]. If swarm behavior is adapted for technical systems, the principle of homogeneity is also often copied for evolutionary designing the respective agents concerning their capabilities, e.g., by using genetic algorithms [Rubenstein et al., 2014; Vásárhelyi et al., 2014; Dorigo et al., 2004]. Because we want to face the challenge we describe in Section 1.3.1 which inevitably requires the system to be able to execute more than one specific task and we want to face the challenge we describe in Section 1.3.6 by applying different swarm behavior for solving tasks where it is appropriate, there are two possible solutions to tackle both challenges at the same time. The first possible solution is to increase the number of agents with different capabilities massively. Because this can only scale up to a certain point because of the proportionally increasing maintenance effort required if

agents are not only simulated but flying robots like UAVs, this solution is not viable. The second possible solution is to allow agents to be heterogeneously configured concerning their capabilities. This requires techniques to guarantee that all agents have the required capabilities if necessary, e.g., to participate in a swarm algorithm despite being configured heterogeneously.

1.4 Scientific Contributions and Overview

To tackle the challenges from Section 1.3, we developed an integrated approach Combining Planning with Self-Organization for handling the problem of Mission Programming for Flying Ensembles. In the following chapters, we present how we achieve this in detail by focusing on the different aspects of our approach. The contributions of this thesis are the result of research published in 15 related and peer-reviewed papers the author of this thesis co-authored.[1] We evaluate our findings by applying them to four case studies from two different problem domains, which we introduce in Chapter 2.

- We introduce the **Application Class of *Search, Continuously Observe, and React (SCORe) Missions* for Ensembles** we describe in Chapter 2. SCORe missions subsume well-known application classes of Environmental Monitoring, Distributed Surveillance, SAR, and Major Catastrophe Handling present in the current research literature. We published this idea in Kosak et al. [2016a].
- We introduce ***Multipotent Systems* as a New System Classification Type** for flying ensembles in Section 3.1. Multipotent Systems base on the idea of strictly separating agents and capabilities during system-design, making them physically reconfigurable at run-time. This enables flying ensembles to profit from the benefits of both system classes currently established in the literature, i.e., systems consisting either of heterogeneously or homogeneously configured agents. If necessary, agents in a Multipotent System can gain scalability and robustness during run-time by self-adapting their configurations to generate a homogeneous system, e.g., for executing goal-oriented swarm algorithms. Otherwise, agents in a Multipotent System can become heterogeneously configured specialists for executing missions requiring such. We published this idea in Kosak [2018]; Kosak et al. [2018]
- We present a ***Layered Reference Architecture* for Multipotent Systems** aiming at flying ensembles for handling SCORe missions in Section 3.2. Implementing this reference architecture enables the autonomous cooperation of multiple agents in flying ensembles to handle complex missions in versatile use cases and applications. Mission execution then can be supervised by a human user. We achieve autonomy and flexibility by integrating appropriate self-organization mechanisms on different layers of this architecture. We published this idea initially in [Kosak et al., 2016a] and extended with respective features in [Kosak, 2017; Kosak et al., 2018; Kosak, 2018; Kosak et al., 2020a,b].
- We provide a ***Prototypical Implementation* of the Reference Architecture for Multipotent Systems** in Section 3.3 to validate its practicability for flying ensembles. We base this implementation on the multi-agent programming framework Jadex Active

[1]The research was partially funded by DFG (German Research Foundation), project COMBO - *Kombination von Planung, Selbst-Organisation und Rekonfiguration in einem Roboterensemble zur Ausführung von SCORe Missionen* - grant number 402956354.

Components [Braubach and Pokahr, 2012] which we integrate with the robotic simulation and visualization environments of the Robotics Application Programming Interface (API) [Angerer et al., 2013] and Robot Operating System (ROS) [Quigley et al., 2009]. We further deployed this prototype to actual robots and present its potential in indoor demonstrations and field experiments, showing its extensibility to driving ensembles. We published the results in [Kosak et al., 2016b, 2018; Kosak, 2018; Kosak et al., 2020a,b].

- We present an **Approach for a *Multi-Agent Script Programming Language for Ensembles* (Maple)** integrating the design and planning of SCORe missions in Chapter 4. With that, we can program complex missions for ensembles on the abstract level of agents' capabilities and capabilities emerging from collectives of agents implementing swarm behavior. We deliver an approach for a graphical user interface for programming such missions declaratively, concentrating on the relevant elements necessary for ensembles and hiding complexity. Our approach, therefore, extends the concepts of Hierarchical Task Networks (HTN). We deliver a prototypical implementation for Maple and showcase its potential by example and comparison with other approaches from the literature. We published the concepts and results in [Kosak et al., 2019] and [Kosak et al., 2020b].
- We provide **Multiple SO-Mechanisms for the *Autonomous Execution of SCORe Missions.*** We show in our evaluations that these mechanisms can be deployed and executed within actual flying robots in the field and laboratory experiments [Kosak et al., 2016b, 2018], as well as in theoretical and statistical analysis [Kosak et al., 2016a; Hanke et al., 2018]. SO-Mechanisms we provide include
  - an **Approach for the *Distributed, Self-Aware, and Market-Based Mechanism for Ensemble Formation (SELF-MADE)*** necessary to form ensembles required in SCORe missions, which we introduce and evaluate in Chapter 5 and published in [Kosak et al., 2016a, 2020b],
  - an **Approach for a *Task and Resource Allocation Strategy for Multi-Agent Systems (TRANSFORMAS)*** enabling agents in the ensemble to self-adapt their physical configuration according to mission requirements we introduce and evaluate in Chapter 6 and published in [Hanke et al., 2018],
  - an **Approach for the *Autonomous Coordination of SCORe Missions*** controlled within the ensemble we introduce and evaluate in Chapter 7 and published in [Kosak et al., 2019, 2020a].
- We provide an **Approach for an *Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE)*** that enables the generic implementation of a general pattern for swarm behavior usable in flying ensembles in Section 7.3. We show that this generalization is appropriate and feasible by adapting different swarm algorithms from literature for that pattern, published in Kosak et al. [2020a]. We further demonstrate the scalability and robustness of this approach within the multi-agent systems simulation environment NetLogo CCL [2020] and its applicability to flying ensembles by integrating it in our prototypical implementation of the reference architecture.
- We provide an **Approach to Achieve *Goal-Oriented Swarm Behavior*** and other collective behavior in Section 7.5. With that, we can quantify the emergent effect of a swarm algorithm's execution and use its result in subsequent instructions for ensembles. In contrast to the current use of swarm behavior in other research, we can use swarm

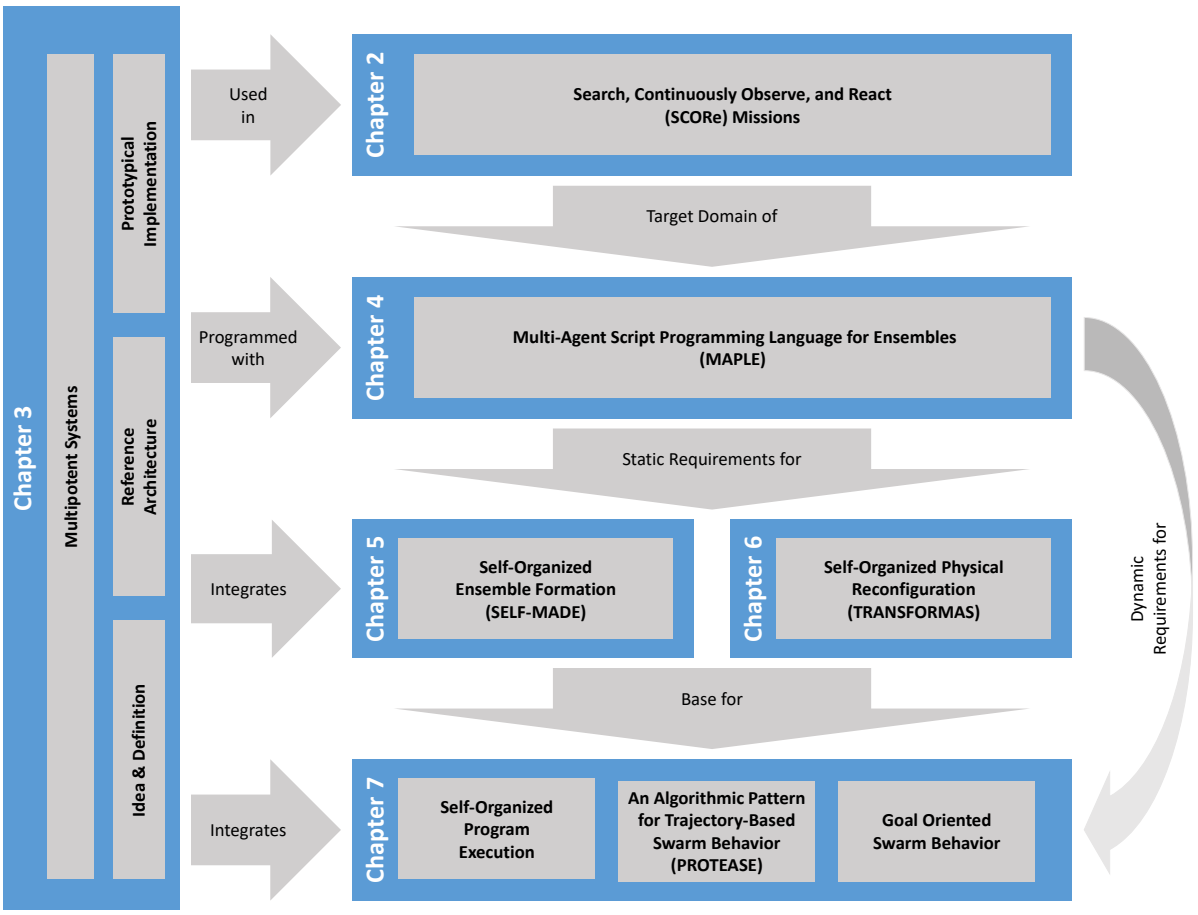


Figure 1.2: Overview on the contents and contributions of this thesis.

behavior in complex missions autonomously and integrate it seamlessly with the execution of other agent capabilities. We show how missions for the different case studies can benefit from swarm and aggregate behavior by example and theoretical analysis and how we integrate it in our prototypical implementation of the reference architecture for Multipotent Systems. We published respective concepts and evaluation in [Kosak et al., 2020a].

- We provide a **Self-Awareness Concept for Achieving *Robust Execution SCORe Missions*** aside from exploiting swarm behavior, if SCORe missions require heterogeneous agents in Section 7.7. We present a theoretical analysis of possible failures in different states of a SCORe mission and provide an observer/controller concept that can direct future work to achieve robustness during the execution of a mission for a flying ensemble. We present preliminary results on the feasibility of that concept in our evaluations.

To conclude this thesis, we summarize our findings in Chapter 8. There, we also give an outlook on possible future research directions enabled by the scientific contributions achieved in this thesis. Figure 1.2 summarizes the contents and contributions of this thesis.

Chapter Summary and Outlook

In this chapter, we motivated the problem of Mission Programming for Flying Ensembles. We highlighted the need for an innovative and integrated approach by analyzing related work concerning the research field. Further, we elaborated on the current challenges we see in developing such an approach and gave an overview of our approach of Combining Planning with Self-Organization we present in this thesis.

Because our approach for Mission Programming for Flying Ensembles aims at being flexible concerning its application to different use cases and application domains, we present such in the following chapter. We will use four use-cases from the two application domains of *Environmental Monitoring* and *Emergency Response in Major Catastrophe Handling* to illustrate the benefits of our approach by example and to prove the feasibility and applicability of the respective concepts integrated into our approach for Mission Programming for Flying Ensembles by Combining Planning with Self-Organization.

Chapter 2

Target Domain and Applications

Summary. In the last chapter, we proposed that use cases for applying Mission Programming for Flying Ensembles can be very versatile. Thus, we search for an approach for Combining Planning with Self-Organization that is flexible enough to deal with that versatility. In this chapter, we state more precisely this versatility by defining the target domain of our approach. Therefore, we introduce the mission class of Search, Continuously Observe, and React (SCORe) missions subsuming many current use cases for ensembles from literature. We give an overview of related work and classify applications according to the SCORe definition. Further, we introduce four different case studies from the field of Environmental Monitoring and Major Catastrophe Handling. We provide examples from these case studies for describing and evaluating the different relevant aspects of our approach in the following chapters.

Publication. Contents of this chapter have been published in [Kosak, 2015; Kosak et al., 2016a,b; Kosak, 2017; Wolf et al., 2017; Hanke et al., 2018; Kosak et al., 2018; Kosak, 2018; Kosak et al., 2019, 2020a,b].

2.1 The Domain of Search, Continuously Observe, and React (SCORe) Missions

Heterogeneous ensembles are becoming increasingly popular as it is appealing to combine the strengths of different devices – especially for *Search and Rescue* missions like proposed in [Scherer et al., 2015; Dominguez et al., 2017; Malaschuk and Dyumin, 2020]. Approaches from that field often use ensembles consisting of multiple agents (mostly UAVs) to simultaneously search in different or spatially large areas, e.g., for reducing the time until missing persons are found. Another application of such flying ensembles is the *observation* of critical infrastructure such as pipelines, railways, or wind turbines [SNCF Réseau, 2015]). Here, the area to cover is predefined, and the ensemble has to observe this known area simultaneously. In each of the scenarios mentioned above, the ensemble has to achieve a small set of predefined tasks. In some cases, even the ensemble's hardware itself has been designed exclusively for a specific task. From our point of view, there is great potential for ensembles far beyond these application areas.

We have identified the class of *Search, Continuously Observe, and React (SCORe)* missions by integrating aspects of the domain of Distributed Surveillance with that of Major Catastrophe Handling, which we consider to be a generalization of SAR missions as they are defined by Murphy et al. [2008]. The following properties characterize SCORe missions:

1. Search one or multiple a priori unknown parameters;
2. Continuously Observe and, if necessary, track the relevant parameters;
3. perform both, S and CO in a potential spatially large area;
4. collectively evaluate gathered data online in the ensemble; and
5. trigger Reactions by user input or due to abnormalities or patterns in evaluated data.

SCORe missions are necessary, e.g., in Major Catastrophe Handling scenarios like chemical or nuclear accidents that often already include search and rescue missions (e.g., searching and rescuing endangered persons in the affected area) but might also require continuous observation of a specific area. Furthermore, SCORe missions come into play in other scenarios where their application is not that obvious. We can find examples, e.g., in Environmental Monitoring where in-situ measurements of climate parameters like temperature, humidity, and greenhouse gas distribution as performed by Wolf et al. [2017] is of particular interest. In that research field, it is often essential to continuously gather consistent data at different locations over a long period, and if possible, detect and further investigate related phenomena as a reaction.

To demonstrate the generality of the problem class of SCORe missions, we now investigate applications from related work and classify them according to the SCORe missions properties we defined beforehand. We then introduce different case studies from the field of SCORe missions in detail that serve as running examples for illustrating and evaluating the different aspects of our approach of Combining Planning with Self-Organization for solving the problem of Mission Programming for Flying Ensembles.

2.2 Related Work

Many current projects investigate ensembles and how ensembles can be applied in different use cases, environments, and applications. Some of them can be classified as complete SCORe missions, i.e., consisting of phases *Search*, *Continuously Observe*, and *React* that characteristic the SCORe class, others only as partial SCORe missions addressing only specific phases of a complete SCORe mission. Because related work on the topic is manifold, we can not provide a complete list of all projects and approaches. Therefore, we discuss some of the most recent and relevant approaches from the use cases of *Environmental Monitoring*, *Distributed Surveillance*, *Search and Rescue (SAR)*, and *Major Catastrophe Handling* categorizing them according to our definition of SCORe missions. We also take into account approaches that are not restricted to flying ensembles consisting of UAV only. In addition, we investigate in those approaches involving Unmanned Underwater Vehicle (UUV), Unmanned Surface Vehicle (USV), or Unmanned Ground Vehicle (UGV) if they are of relevance for us, e.g., because they also apply swarm behavior to one or more phases of a SCORe mission or they include autonomous mission progress between two or more phases of a SCORe mission.

2.2.1 Environmental Monitoring

If ensembles are applied for the use case of Environmental Monitoring, we can most often find only partial SCORe missions. As the main goal in Environmental Monitoring is to collect data

in spatially large environments, the usage of Swarm-Robotic Systems is common because these systems provide valuable properties like scalability and robustness for the use case. Because most approaches from Swarm-Robotic Systems rely on additional guidance if interdependent decisions or actions must be made after the Swarm-Robotic Systems has achieved the goal it is trained for [Brambilla et al., 2013] we see only a few approaches in this domain handling complete SCORe missions.

In the project NAMOS [Caron et al., 2009], e.g., distributed Environmental Monitoring using mobile Unmanned Surface Vehicle (USV)s and UAVs is performed. The ensemble in NAMOS is used for in-situ measurements monitoring the growth and distribution of specific types of plankton. Thus, ensembles in NAMOS execute an instance of Continuously Observe. Because the evaluation of derived data is only performed offline, i.e., not within the ensemble, there is no subsequent task for ensembles in NAMOS that might arise from such an evaluation, i.e., there is no React phase. Together with the fact that no phase for autonomous Search is contained, we can classify NAMOS only to address partial SCORe missions. The same holds for the project CORATAM [Duarte et al., 2016]. There, USVs are used to monitor an area of interest using onboard sensors, i.e., ensembles in CORATAM execute an instance of Continuously Observe. While the USV can perform multiple different swarm behaviors for moving in that area and also combines this behavior to a scripted mission, there is no autonomous evaluation of data resulting from the monitoring, i.e., ensembles in CORATAM do not execute any React phase. Because missions in CORATAM do not contain any Search phase, we can classify them only to address partial SCORe missions. However, we can also see some projects in Environmental Monitoring embedding complete SCORe missions in their approaches. In the project CoCoRo [Schmickl et al., 2011] and its subsequent project subCULTron [Thenius et al., 2016] autonomous ensembles of UUVs and USVs use techniques from the Swarm-Robotic community for executing different types of tasks embedded in a static mission. Ensembles in CoCoRo and subCULTron execute, e.g., search and explore tasks (i.e., instances of Search), ecological monitoring tasks (i.e., an instance of Continuously Observe), maintenance tasks, and harvesting tasks (instances of React) in an underwater environment. Thus, CoCoRo and subCULTron handle full, but very specific SCORe mission without providing the flexibility to easy adapt once designed missions.

2.2.2 Distributed Surveillance

When focusing on the use case of Distributed Surveillance, e.g., for surveying critical infrastructure or areas of interest in general, we can mainly find applications containing only partial SCORe missions. Approaches involving ensembles in this domain typically only handle the part of Continuously Observe without processing derived results within complex SCORe missions. Saska et al. [2016], e.g., focus on optimizing the covering of a specific area of interest using an ensemble of UAVs but do not further investigate in subsequent tasks, i.e., they do not take into account the Search nor the React phase of a SCORe mission. Also, Stolfi et al. [2020] only focus on approaches for maximizing the coverage in a Distributed Surveillance scenario by using an ensemble of UAVs and thus limit their approach to the Continuously Observe phase of a SCORe mission. The same holds for the research of Liu et al. [2020]. While their approach for Distributed Surveillance of an urban area includes different applications of the Continuously Observe phase of a SCORe mission executed by an ensemble of UAVs, they do not consider any other phases of SCORe missions.

2.2.3 Search and Rescue

In the domain of Search and Rescue (SAR), approaches dealing with full SCORe missions involving all phases are rare, and we can find many applications of ensembles used for partial SCORe missions instead. This holds even more if approaches involve swarm behavior for dealing with SAR.

With their approach in [Yang et al., 2017], the authors adopt an ant colony algorithm to be applied by an ensemble of UAVs for improving the efficiency of searching objects in an uncertain environment. Because their approach does not include any further autonomous action, we classify it as a partial SCORe mission, only involving the Search phase. While Ruetten et al. [2020] state that their approach is dedicated to SAR in general, they only focus on algorithms for optimizing the search for objects of interest in a given area with an ensemble of UAVs. Thus, the approach in [Ruetten et al., 2020] handles a partial SCORe mission only. Gade and Joshi [2013] focus on handling the Search part of a SCORe mission by applying swarm behavior to an ensemble of UAVs but do not involve subsequent autonomous actions. Thus, we also classify the approach of [Gade and Joshi, 2013] to be an approach handling a partial SCORe mission only.

In the context of SAR, only a few approaches explicitly handle the React part of a SCORe mission and use swarm behavior. One application dealing with the relevant topics in this field is that of collective transport. In the approach of Mondada et al. [2005], an ensemble of UGVs can cooperatively rescue an endangered person from a hazardous area. Because the approach does not involve autonomous actions for Search and Continuously Observe, we classify it to handle a partial SCORe mission only. The same holds for the approach of Wilson et al. [2014], who also dedicate their approach to be applicable in SAR. By following the stigmergic rules adapted from ants in nature, a not further defined ensemble can cooperatively form up around objects that need to be transported. While their approach can be applied to a React phase of a SCORe mission, Search and Continuously Observe phases are neglected. Thus, they can handle partial SCORe missions only.

Nevertheless, we can also find projects that handle complete SCORe missions. In the project Swarmanoid [Dorigo et al., 2013], multiple swarm behaviors are applied to an ensemble consisting of UAVs and UGVs. In a complete SCORe mission, Dorigo et al. [2013] first use UAVs for searching an object of interest, i.e., the ensemble performs the Search part of a SCORe mission. Second, the same UAV ensemble observes a path towards the object, i.e., executes the Continuously Observe part of a SCORe mission. Third, the UGVs collect the object of interest, i.e., execute the React phase of a SCORe mission. However, like in subCULTron from the use case of Environmental Monitoring, this SCORe mission is precisely scripted before, requiring a high engineering effort and making it inflexible for application in other use cases.

Other approaches not focusing on Swarm-Robotic Systems include autonomous interconnection of multiple phases of a SCORe mission. In the project sherpa [Marconi et al., 2013], ensembles consisting of UAVs and UGVs are used to support rescue forces in a SAR scenario located in an alpine environment. UAVs are used to search and identify endangered persons that require help, i.e., execute an instance of the Search part from a SCORe mission. UGVs support human users while rescuing those persons by transporting heavy equipment towards the detected persons, i.e., execute the React phase of a SCORe mission. There is no explicitly defined Continuously Observe part in the approach of Marconi et al. [2013], so we can classify the missions executed in sherpa to be partial SCORe missions only. In the project swarmix [Flushing et al., 2014], human capabilities are combined with those of animals (e.g., dogs) and

artificial agents (e.g., UAVs) to create heterogeneous ensembles for handling SAR missions. Because ensembles consisting of UAVs are only used for Search and Continuously Observe while humans and animals handle the React phase, we classify the approach of SWARMIX as partial SCORe mission.

2.2.4 Major Catastrophe Handling

An extension to applications in SAR is that of applications aiming at Major Catastrophe Handling, e.g., necessary in case of chemical accidents, nuclear accidents, earthquakes, floods, or wilderness/forest fires Restas et al. [2015]. While the scenario calls for applying complete SCORe missions (cf. our case study in Section 2.5 and Section 2.6), little research have been performed concerning the relevant parts.

In the project AirShield, Daniel et al. [2009] aim at handling chemical accident scenarios. In these scenarios, they focus on establishing stable wireless connections autonomously over long distances using multi-UAV systems to compensate for possibly insufficient public communication networks. They aim to stream data from an area of interest to a distant operator for further evaluation. This evaluation and an appropriate reaction have to be determined by the human user of the system. Thus, we can classify them as a partial SCORe mission focusing on the Continuously Observe phase only.

In the project OSIRIS, Lewyckyj et al. [2007] focus establishing a UAV for monitoring forest fires from high altitudes. In their scenario, a single UAV is controlled manually for generating an overview on the situation producing data to decide on the correct actions of rescue forces in different situations relevant for Major Catastrophe Handling, e.g., preparing recovery. While the system used is only controlled in a semi-autonomous manner and the application aims primarily at the Continuously Observe phase of a SCORe mission, the OSIRIS nevertheless shows the need for supporting rescue forces with autonomous UAV systems.

In [Roldán et al., 2015] focus on establishing appropriate measures for modeling and planning multi-UAV missions used for Major Catastrophe Handling. By enriching their approach with possibilities for controlling and monitoring the respective state of the system during the execution of such missions, Roldán et al. [2015] aim at providing operators adequate information for deciding on the correct measures for intervention. Their studies try to determine an appropriate methodology for modeling different relevant tasks that can occur when facing major catastrophe scenarios like searching for suspects, monitoring fires, and supporting rescue forces. While the proposed approach for using Petri nets or hidden Markov models is appealing because of its clarity, it nevertheless works on a very high level of abstraction. Roldán et al. [2015] focus all possible phases of SCORe missions isolated but do not provide appropriate solutions for interconnecting different phases on the level of detail necessary for applying real hardware systems.

Zhu et al. [2019] aim at using multi-UAV in post-earthquake scenarios. They propose an approach for the rapid assignment of assessment tasks in such scenarios for generating an as-soon-as-possible overview on the affected area and support detected victims with needed resources. We can classify this as the Search and the React phases of a SCORe mission. Because their system is intended to deliver generated data to human coordinators, Zhu et al. [2019] interconnecting the different phases of SCORe missions autonomously within their approach.

In [Ghamry et al., 2017] a forest fire scenario is handled by an autonomous multi-UAV system. The approach presented focuses on the fire extinguishing task only, assuming positions of fires are already known. Therefore, Ghamry et al. [2017] present an auction-based task

allocation mechanism for assigning fire positions to individual UAV. Because the approach does not involve any other phases of a SCORe mission, we can classify it as a partial SCORe mission focusing on the React phase only.

Sudhakar et al. [2020] in contrast, focus the detection and monitoring task in such forest fire scenarios, i.e., the Search and Continuously Observe phases of a SCORe mission. While their approach aims at reliable detection and the avoidance of false alerts, it does not aim to use this data to react to correctly detected fire events. Thus, Sudhakar et al. [2020] handle a partial SCORe mission.

2.3 Case Study 1: Improving Climate Models in Environmental Monitoring

A case study where flying ensembles are executing complete and partial SCORe missions can be beneficially applied, which we want to further analyze in this thesis, stems from the domain of Environmental Monitoring. Flying ensembles can validate and improve regional climate models by measuring and monitoring relevant environmental parameters. With their possibility to evaluate measured data online, they also offer the possibility to investigate interesting phenomena occurring during measurements, including appropriate reactions. This enables a more precise analysis of environmental processes that potentially can impact health and living conditions. The following examples are motivated and directed by the results of multiple workshops performed in cooperation with researchers of the *Division of Atmospheric Environmental Research (IFU), Institute of Meteorology and Climate Research (IMK), Karlsruhe Institute of Technology, Garmisch-Partenkirchen, Germany* and the *Institute of Geography (IGUA), University of Augsburg, Germany* during the preparation and post-processing of the Scale Crossing Intensive Research Campaigns (ScaleX) field experiments 2015 and 2016 [Zeeman, 2019] during October 2014 and March 2017.

One parameter considered in Environmental Monitoring that is of particular interest for human health is that of the Particulate Matter (PM) concentration, e.g., measured in *parts per million (ppm)*, within the lowest Atmospheric Boundary Layer (ABL). According to Davidson et al. [2005], PM characteristics "are believed to influence human health risks" and the "health effects of PM are thought to be strongly associated with particle size, composition, and concentration". PM can "cause a wide range of diseases that lead to a significant reduction of human life". Especially persons having health dispositions concerning air quality like asthma thus are interested in PM predictions in their daily lives [Lin et al., 2002]. Currently, predictions concerning the concentration of PM rely on meteorologic models taking into account a series of different data (e.g., latest stationary measurements, meteorological data, and other geographic data) [Greenpeace, 2006]. For deriving data on a large scale, models are typically structured in uniform grids of different scales. The EURAD-IM system [RIU, 2018], e.g., models grids at a macro-scale of 125 kilometers, meso-scale of 25 kilometers, and a micro scale of 5 kilometers (cf. Figure 2.1). The relatively high grid length on a micro-scale and the high effort for updating models with live measurements currently restrict the usefulness of real-time predictions for air quality [Jakobs et al., 2002]. Furthermore, according to Wolf et al. [2017], "the mismatch between observations of land surface processes and their modeled equivalents is still so large that it constitutes a major source of uncertainty in climate models". Thus, there is a general need to enrich and cross-validate models using in-situ measurements concerning the parameters of interest, e.g., for achieving better forecasts.

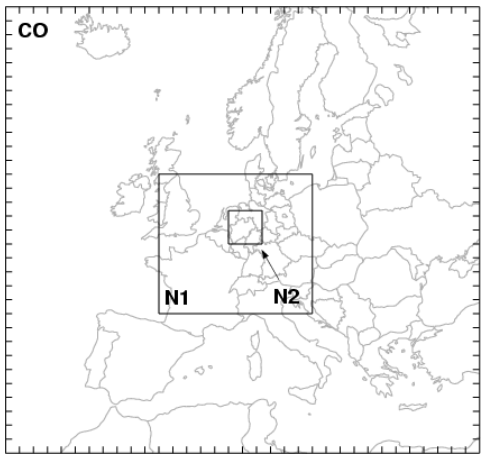


Figure 2.1: The grid based model used in the EURAD-IM system [RIU, 2018], showing macro-scale (C0), meso-scale (N1), and micro-scale (N2) of the model.

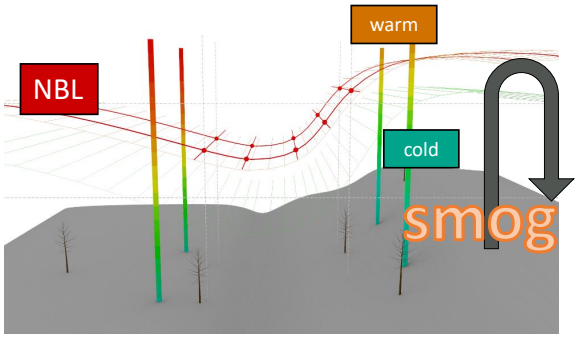
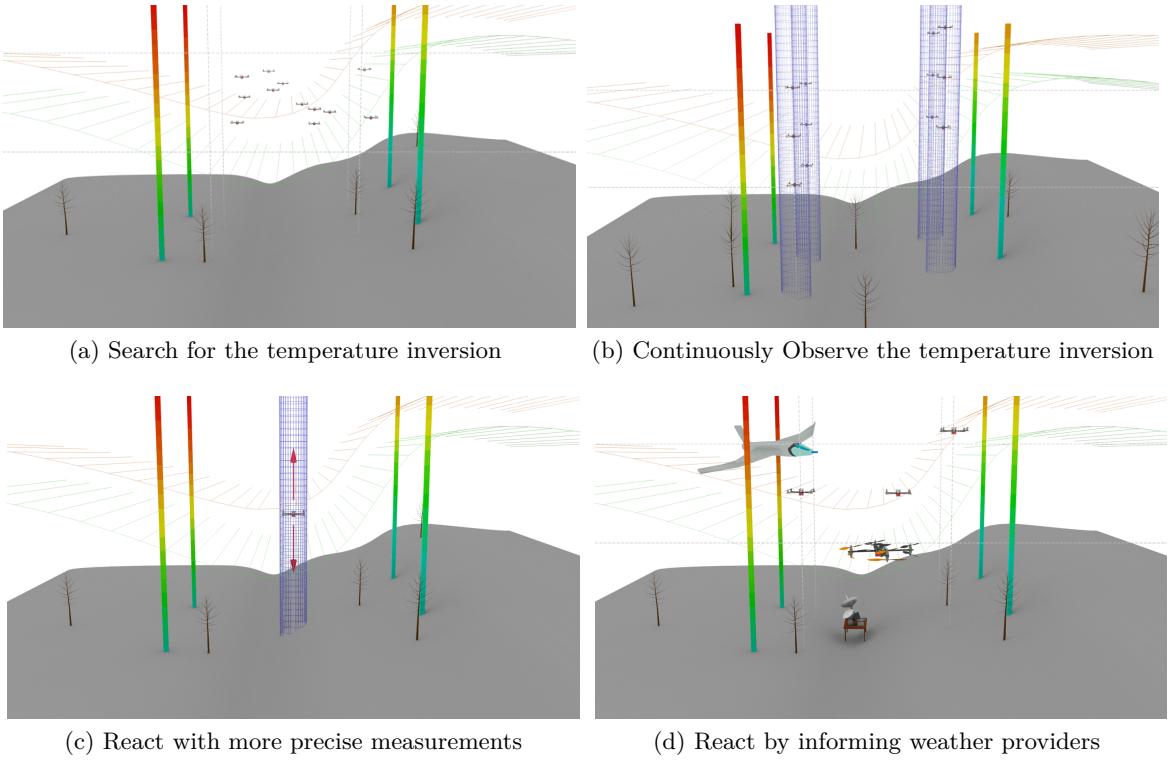


Figure 2.2: Temperature inversion in the NBL. The default flux of aerosols (e.g., smog including PM) from lower (cooler) into higher (warmer) layers of the atmosphere is impeded so that, according to [Trinh et al., 2019], critical concentrations can accumulate in the lower boundary layer.

Furthermore, local phenomena are affecting relevant parameters, e.g., PM concentration, that cannot be modeled precisely enough at all. One of those phenomena is the occurrence, distribution, and location of a temperature inversion in the NBL. During the night, air conditions in the NBL are often dominated by the inversion of the typical temperature gradient from *warm at ground level* to *cold in higher regions*: Due to geographical conditions (complex terrain) and physical effects (faster cooling down of the ground than the upper air layers) in addition to meteorological phenomena (wind, precipitation) during the night, this typical gradient can be inverted at a certain height above ground. The height where this inversion happens and its local expansion is relevant for approximating the air quality concerning the current situation as well as for the following day [Trinh et al., 2019]: Smog can accumulate below the inversion layer which can influence air quality massively [Pöschl, 2005] (cf. Figure 2.2). The crucial task in this setting is to find and observe the time and location where the inversion happens, determine its distribution, and monitor how long it lasts until the typical gradient gets re-established. The benefits of using a flying ensemble in that scenario are mani-



(a) Search for the temperature inversion

(b) Continuously Observe the temperature inversion

(c) React with more precise measurements

(d) React by informing weather providers

Figure 2.3: Flying ensemble executing a SCORe mission for detecting the meteorological phenomena of a temperature inversion and possible consequences for human health.

fold. While we can use the flying ensemble foremost for detecting the location of the NBL, i.e., we can determine the height where the inversion of temperature starts, we can then also let the ensemble monitor the local conditions (e.g., wind, PM concentrations, humidity) at this position and determine the spatial distribution of the inversion layer. Further, we can deploy a flying ensemble flexibly at almost any geographical position we need because of its mobility. This is an improvement in comparison to the current state of in-situ monitoring, which is mainly achieved by stationary measuring units, e.g., measuring towers [Kaimal and Finnigan, 1994]. While there are also other mobile techniques for coming by the drawback of stationary measuring units, e.g., based on 3D Doppler boundary layer lidar (cf. Figure 2.4), those techniques have other disadvantages. We further elaborate on these in Section 2.4 and illustrate how flying ensembles can support improving measurements also in these cases. Moreover, a flying ensemble can perform measurements at different relevant positions time-synchronously. On the one hand, we can use this to understand better the spatial expansion of local meteorological phenomena (e.g., the structure of the temperature inversion layer). On the other hand, we can exploit collectively performed measurements for canceling out the measurement errors that lightweight sensors are prone to, which are known to be a pitfall when using single UAV systems only [Sensirion, 2018].

Possible SCORe Mission for Flying Ensembles

We can formulate the mission of detecting and evaluating the influence of the temperature inversion in the NBL as a sequence of Search and Continuously Observe phases in the context of our SCORe pattern:

- At first, we need the flying ensemble to execute the Search phase of a SCORe mission to determine whether there exists a local temperature inversion under the respective environmental and meteorological conditions. Therefore, we require as many UAVs from the ensemble as possible to execute temperature measurements while ascending from ground level up to the relevant height Above Ground Level (AGL) for searching the height and spatial distribution of the inversion layer (cf. Figure 2.3a).
- After identifying the properties of the temperature inversion, the flying ensemble can monitor the spatial distribution of the temperature inversion while measuring parameters of relevance, e.g., PM concentration and wind conditions, at the same time (cf. Figure 2.3b). Thus, the flying ensemble executes the Continuously Observe phase of a SCORe mission.
- When the concentration reaches a level that can harm human health, the flying ensemble can perform appropriate actions. Because currently no direct countermeasures are reducing already emitted PM, in the React phase of a SCORe mission, a flying ensemble can only help indirectly in this case. Possible measures can contain, e.g., informing local weather service providers to update their health forecasts (cf. Figure 2.7) or further investigations in the phenomena with more precise sensors (cf. Figure 2.3c).

Using flying ensembles to improve existing climate models with local and up-to-date measurements can enable much more precise forecasts in the future. This improvement can help predict developments of meteorological parameters that can have a severe impact on human health and are currently derived on a too coarse level.

2.4 Case Study 2: Innovative Measurement Methods in Environmental Monitoring

Another case study from the domain of Environmental Monitoring related to the one we illustrate in Section 2.3 we are interested in in this thesis focuses on the possibilities flying ensembles offer for verifying and complementing the measurements performed by remote sensing systems. Because according to Wolf et al. [2017], "Battery-operated UAVs have no exhaust, and very low heat emissions, and can [...] hold a given position even in convectively turbulent conditions." we can also use flying ensembles for verifying and complementing the measurements of remote sensing techniques, e.g., the 3D Doppler boundary layer lidar system (cf. Figure 2.4). This is necessary as the 3D Doppler boundary layer lidar system is restricted in the range of possible measurements in specific environments, e.g., when facing obstacles like hills, trees, buildings. In such environments, the 3D Doppler boundary layer lidar technique currently can not handle close-to-ground measurement. Further, similar to stationary measurement towers that can only provide measurements at fixed heights (i.e., at those positions where senors are installed), 3D Doppler boundary layer lidar can only provide reliable measurements with a predefined resolution. The system installed during the ScaleX field experiments [Wolf et al., 2017] describe, e.g., can provide measurements only in altitude increments of 18 meters.

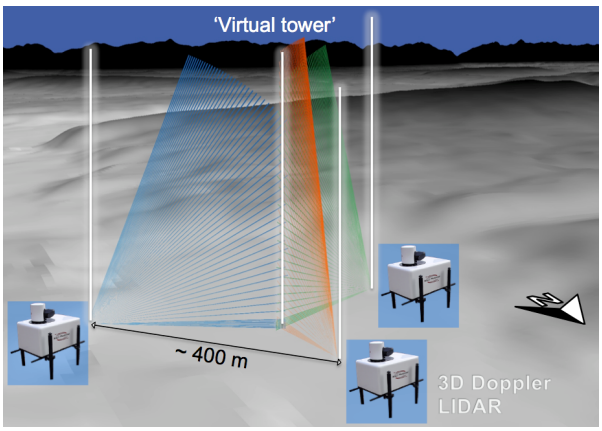
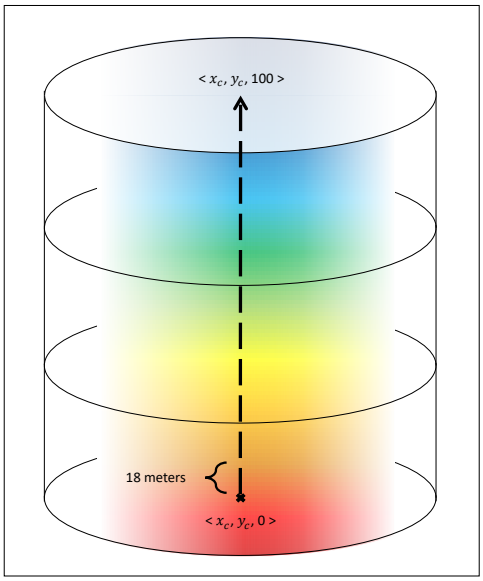


Figure 2.4: The 3D Doppler boundary layer lidar technique can be used for generating virtual meteorological measurement towers for remote temperature and wind measurements, e.g., during ScaleX [Wolf et al., 2017]. Here, three 3D Doppler lidar building a virtual meteorological measurement tower.

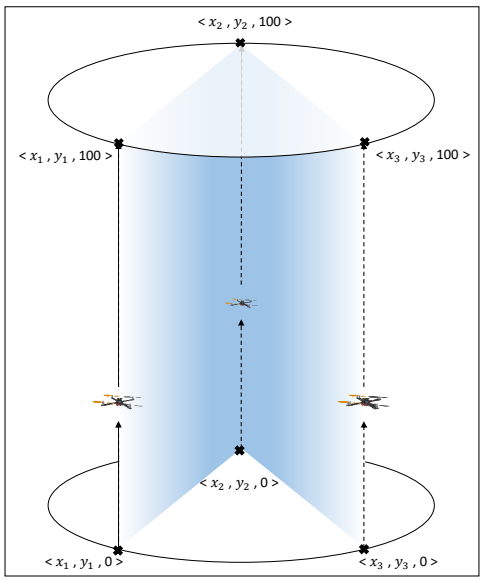
On the contrary, a flying ensemble has no trouble providing measurements in the relevant range (estimations suggest that the DJI Phantom 2 Vision [SZ DJI Technology Co., 2021], e.g., can reach every height between 0 - 2000 m AGL [Brunner, 2018]) and thus can come by the drawback of specific remote sensing systems and complement their measurements. For verifying and complementing the estimated measurements of a virtual measurement tower (cf. Figure 2.5a), we can let the flying ensemble perform coordinated flights at the positions where the 3D Doppler boundary layer lidar devices are located while generating in-situ measurements with lightweight onboard sensors (cf.Figure 2.5b). Further, we can enable flying ensembles to establish other innovative mobile measurement methods to verify and complement remote sensing systems with measurement devices currently only applicable in fixed installations. One example of such a measuring device is that of DTS using long-range fiber-optic cables for achieving temperature measurements at a large scale [Zeeman et al., 2015]. With DTS, we can measure the temperature every meter [Sensornet, 2018a], i.e., with a very fine-grained spatial resolution when compared with remote sensing systems like the 33D Doppler boundary layer lidar system. While DTS systems are mainly used in fixed installations [Zeeman et al., 2015], there are already some approaches for using DTS also in the context of mobile robotics, i.e., UAV [Higgins et al., 2018]. Because at the moment, these systems only use single UAV and not ensembles of them, possibilities for their application are restricted. With a flying ensemble collectively transporting the DTS, we can perform complex flight patterns generating in-situ measured, large scale temperature profiles at flexible locations, e.g., for complementing and verifying the measurements of the 3D Doppler boundary layer lidar system like proposed in Figures 2.5c and 2.5d.

Possible SCORe Mission for Flying Ensembles

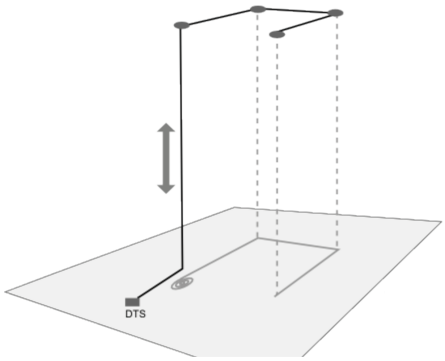
We can formulate partial SCORe missions using the flying ensemble as one large-scale measuring unit for collecting relevant data. With such, we can help to verify and complementing measurements of remote sensing devices. For example, we can derive measurements comparable to that of the proposed 3D Doppler boundary layer lidar technique in [Wolf et al.,



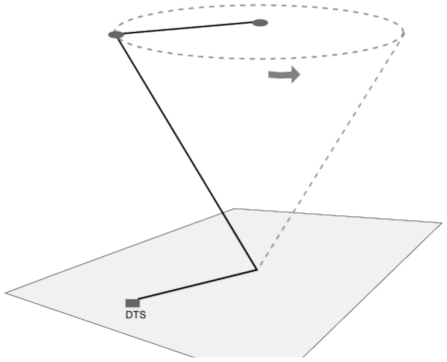
(a) estimated temperature profile



(b) flight pattern at lidar positions



(c) flight pattern using DTS



(d) flight pattern using DTS

Figure 2.5: We depict an estimated temperature profile as generated by 3D Doppler boundary layer lidar remote sensing instruments used by Wolf et al. [2017] in Figure 2.5a. In Figure 2.5b, we show a flight pattern of a flying ensemble we can use to verify and complement these virtual profiles with respective in-situ measurements using lightweight onboard sensors. Figures 2.5c and 2.5d demonstrate how we can perform this in-situ validation using a flying ensemble to collectively transport a DTS measuring device within different patterns.

2017]. Therefore, we can define a partial SCORe mission to collect data with a flying ensemble carrying lightweight onboard sensors:

- Continuously Observe temperature and wind close to the position of the virtual measuring tower generated by the 3D Doppler boundary layer lidar system by collecting the relevant data for post-mission evaluation with the flying ensemble.
  - We can achieve this, e.g., by commanding a flying ensemble to collectively create respective measurements at the same altitude while ascending and descending in a relevant range in a coordinated flight pattern (cf. Figure 2.5b).

– Alternatively, we can achieve the respective measurements by commanding the flying ensemble to collectively transport a DTS measuring system while executing complex flight patterns, "scanning" the current temperature by moving the DTS along a predefined path (cf. Figures 2.5c and 2.5d).

After executing this partial SCORe mission, we can evaluate measurements performed by the flying ensemble and compare them to those the respective remote sensing system provides. Combining measurements from both the remote sensing system and the flying ensemble can achieve much more precise and complete measurement profiles of the parameters of interest. Thereby, a flying ensemble can play an essential part in developing and establishing new mobile measuring methods providing reliable data where traditional measuring methods like measuring towers are not available.

2.5 Case Study 3: Dealing with Gas Accidents in Major Catastrophes Handling

Another case study we investigate in this thesis because flying ensembles can be beneficially applied there is that of Major Catastrophe Handling required, e.g., in chemical accidents. Unfortunately, such accidents happen quite often as current incidents at BASF in Germany (2016) [CNBC, 2018a], at Arkema in Texas (2017) [CNBC, 2018b], Bayernoil in Germany (2018) [Euronews, 2018], or most recently in a plastics plant from LG Polymers in Andhra Pradesh in India (2020) [The Gurdian, 2020] demonstrate.

The following description of typical handling of chemical accidents performed by firefighters and other rescue forces results from multiple interviews performed with the former longtime head of fire department Augsburg, Frank Habermeier. We combined these with other practical insights and lessons learned we can find within the Major Accident Reporting System (eMARS) of the European Commission [European Commission, 2017]: When handling chemical accidents, firefighters often face the threat of toxic gases, e.g., in road or train accidents or accidents happening in synthetic material plants. This comes with the tasks that urgently need to be accomplished in case of such accidents. Firefighters need to analyze their impact and try to reduce its dangerousness for residents. First of all, firefighters need to clarify the situation by searching the relevant parameter of the respective accident (cf. (1) in Figure 2.6). They need to identify the gas with the highest risk potential, i.e., the conductive gas and its primary source of exposition. After this identification, i.e., when the firefighters know the conductive gas, they need to continuously observe the dissemination of the gas (cf. (2) in Figure 2.6). According to these observations, firefighters then need to generate estimations on the harmfulness of the gas, e.g., whether it disseminates towards a critical infrastructure or endangers residents. The observation can result in the need for appropriate reactions, e.g., the evacuation of threatened inhabited areas (cf. (3) in Figure 2.6).

Currently, firefighters must make observations manually and take actions with limited, incomplete, or even wrong information [Daniel et al., 2009]. Because measurements currently need to be performed by hand, the available staff is often limited, or terrain is challenging to walk through, emergency forces perform measurements of gas concentrations at few locations only [European Commission, 2017]. For estimating the dissemination of the gas, measurements are performed mainly at the location where firefighters assume the accident most likely happened and at a few other points close to the ground around the area. This is problematic because firefighters need to move into endangered areas, possibly exposing themselves to

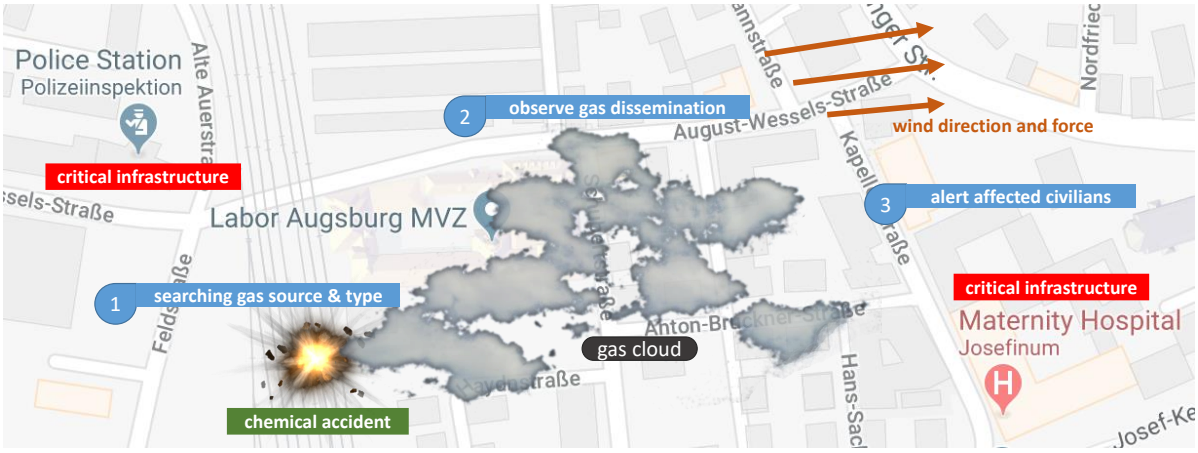


Figure 2.6: Hypothetical gas accident on a railway station in Augsburg, Germany. The wind direction and force influence the gas clouds' spreading range to the east. Dependent on the spreading range, only the eastern critical infrastructure (Maternity Hospital) is affected by the toxic gas cloud. The three steps illustrate the typical sequential procedure for firefighters to handle gas accidents.

the hazardous gases [European Commission, 2017]. Moreover, it is difficult to get a complete overview of the affected area by evaluating these few local observations. Despite these unfavorable circumstances, firefighters must rely on these few measurements to identify the conductive gas and its concentration and estimate the dimensions of the gas cloud. Using such rough estimations that are influenced by too few ground measurements, different expansion patterns of gases, or possibly old weather data, the operational forces have to make far-reaching decisions on whether to evacuate nearby areas, e.g., residential houses, retirement homes, or even hospitals (cf. Figure 2.6). To evacuate a particular area, a typical method for firefighters is to traverse inhabited streets, making loudspeaker announcements to the population. Staff availability permitting, they ring doorbells, particularly in large buildings. Still, it is not easy to reach everyone in rural areas or urban recreation areas. An appropriate technique for getting an overview in such situations is missing, e.g., for detecting people out for walks.

Possible SCORe Mission for Flying Ensembles

A flying ensemble could substantially improve the way such accidents can be handled, as shown by previous work [Daniel et al., 2009], which already considered the possibility of gas cloud tracking using an ensemble consisting of UAVs. Figure 2.7 shows how we envision a flying ensemble executing the relevant mission. We assume that, in general, firefighters will have only a restricted set of hardware available due to the costs and the resulting maintenance efforts. In our example in Figure 2.7, the flying ensemble consists of a set of UAVs and a variety of additional components, like sensors, cameras, and loudspeakers. As illustrated for the chemical gas accident example above, a flying ensemble needs to execute different tasks sequentially or in parallel. Moreover, there are dependencies between tasks and different requirements for the flying ensemble to fulfill all tasks successfully. We can classify those tasks according to our definition of SCORe missions from Section 2.1:

- For identifying the conducting gas and searching for its main source of exposition, i.e.,

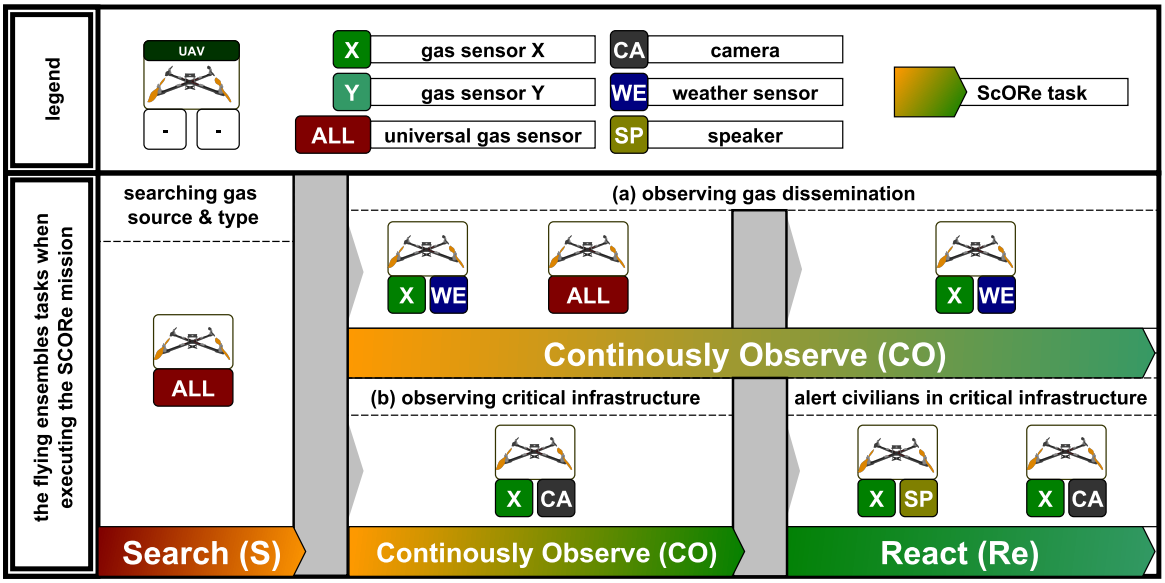


Figure 2.7: Simplified example of how flying ensembles can support the firefighters to deal with the Major Catastrophe case of a chemical accident. The top part of the figure shows the available components (robots as well as additional sensors and actuators) and serves as a legend. The bottom part shows the different phases of a SCORe mission that are necessary and gives an idea of the respectively needed S&A. Dependencies between tasks are ordered from left to right, including parallel execution.

handling the Search phase of a SCORe mission (cf. Figure 2.7), we need to equip the flying ensemble with appropriate multi-gas sensors that can measure the relevant range of candidate gases. Due to the size and weight of these sensors, each UAV might already be fully loaded with this type of sensor.

- When the conducting gas is identified and its source is detected, the tasks for the flying ensemble need is twofold: On the one hand *(a)*, the flying ensemble needs to observe the dissemination of the toxic gas, and on the other hand *(b)*, it needs to particularly monitor critical infrastructures in the vicinity, e.g., a hospital. Both tasks, *(a)* and *(b)*, are instances of the Continuously Observe phase of a SCORe mission. The gas dissemination in (a) can be observed by focusing mainly on the conducting gas (type 'X' in Figure 2.7), for which smaller and lighter sensors might be available. Additional measurements of the current and local weather conditions (temperature, humidity, and wind) with additional lightweight sensors can predict the future expansion of the gas cloud. However, it is still essential to have a universal gas sensor at hand to be able to detect unforeseen additional gases [European Commission, 2017]. To monitor critical infrastructures in *(b)*, at least a sensor for the known predominant gas is necessary. Additionally, cameras can help firefighters get an impression of the situation at the respective sites, e.g., regarding the density and movement of persons.
- If the gas cloud expands to populated areas, evacuations have to be started, and the flying ensemble autonomously can begin to support the evacuation of affected areas, i.e., start the React phase of a SCORe mission. Therefore, the flying ensemble needs

to be equipped with cameras, loudspeakers, or signals to warn people. The speakers help inform people about the evacuation quickly, particularly outside high-rise buildings that firefighters otherwise would have to walk through. Cameras can still be essential to provide an aerial overview of the situation.

In addition to these different task requirements, the flying ensemble may face uncertainties when dealing with chemical accidents like the one described here. Examples of uncertainties are defects of robots at run-time, lack of clarity regarding the initial (environmental) setting the ensemble has to work in, and its development during run-time. These uncertainties make it hard to calculate a complete plan for each member of the flying ensemble in advance. Like any SCORe missions, the example of a chemical accident call for adaptation at run-time. With our approach for Mission Programming for Flying Ensembles by Combining Planning with Self-Organization, we can aim to improve the current state of handling Major Catastrophe Handling scenarios like chemical accidents.

2.6 Case Study 4: Dealing with Forest Fires in Major Catastrophe Handling

Another very relevant use case for the application of a flying ensemble is that of handling forest fires. Losing forests to fires has multiple harmful impacts on society and thus should be urgently avoided. On the one hand, forests play an essential stabilizing role in global climate change and air pollution. Urban forests, e.g., are known to have a positive effect on air quality and can remove air pollution up to a certain degree [Escobedo and Nowak, 2009]. On the other hand, if forest fires occur in high frequency or at a large scale, the burning trees themselves cause air pollution. This can even have a direct and significant influence on human health, as Sastry [2002] had shown by analyzing the effect of a series of forest fires on mortality rates in Indonesia and Malaysia. Of course, forest fires do have not only a critical impact on human health but also on wild animals by destroying their natural habitat [Kolarić et al., 2008]. Unfortunately, forest fires are no rare event today, and big fires destroy millions of hectares of forest every year [Martinez-de Dios et al., 2008]. For many years, these facts were a great motivation for state institutions and researchers to investigate possibilities for avoiding forest fires by using appropriate forest observation technologies or apply countermeasures as soon as possible but also for technologies and measures to fight forest fires actively, if they are present [Yuan et al., 2015; Roldán-Gómez et al., 2021].

Today, if forest fires are not directly detected by the public, e.g., forest workers or civilians, forest observation and fire detection are mainly performed by manned aircraft or satellites. While using manned aircraft for this purpose is very reliable and precise, it typically is very cost intense, requires pilots with high expertise, and can even endanger pilots [Yuan et al., 2015]. Satellite observation of forest fires has drawbacks concerning time scale and spatial resolution. It often has no appropriate response time or can not locate relevant positions precisely enough [San-Miguel-Ayanz and Ravail, 2005]. According to Restás [2014] and Roldán-Gómez et al. [2021] who analyze possible use cases for UAVs for supporting forest fire management, single UAV and ensembles of them can significantly improve on this current state. By their high flexibility in movement and application for different tasks, combined with their relatively low costs compared to using manned aircraft missions, applying UAVs reduces the risk firefighters are exposed to currently.

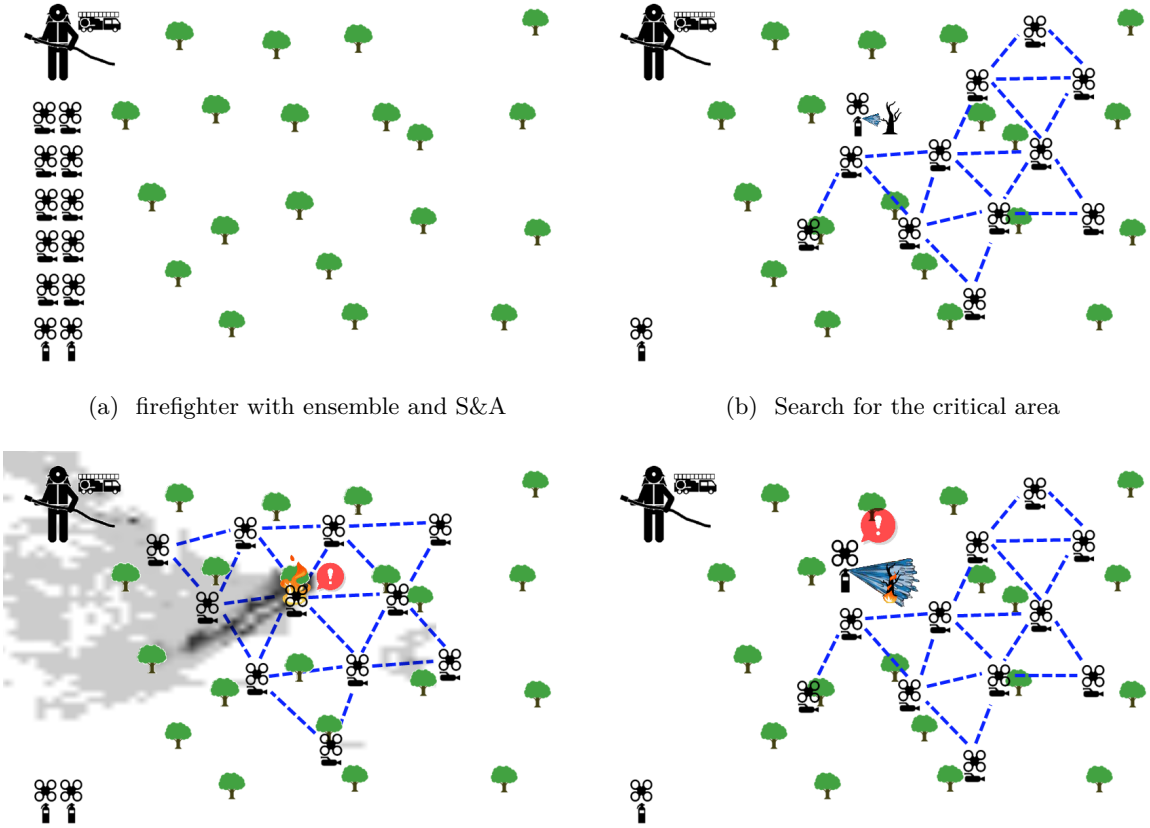
The relevant tasks for an autonomous flying ensemble in fighting against forest fires thus are preventing fires, observing fires, and extinguishing fires if possible [Roldán-Gómez et al., 2021]. According to our definition of SCORe missions in Section 2.1, we can identify the case study of fighting against forest fires to be a full SCORe mission. Detecting the occurrence of a forest fire by actively searching for critical events, e.g., smoke formations, maps to the Search phase of a SCORe mission. The surveillance of a detected fire and its surrounding area, e.g., for detecting subsequent fires that might ignite, maps to the Continuously Observe phase of a SCORe mission. Extinguishing detected fires autonomously if still possible, e.g., by exposing appropriate extinguishing substances or at least informing firefighters accordingly maps to the React phase of a SCORe mission. Related approaches we analyzed in Section 2.2 already show the general feasibility of achieving the Search and Continuously Observe phase of a SCORe mission technically by applying appropriately configured and instructed UAVs or even flying ensembles using swarm behavior. However, also for the most critical part in this case study, the React phase, there already exist technical solutions involving UAV. Because fighting against massive forest fires also requires extinguishing materials scaled accordingly, which often can only be achieved by manned aircraft like the CL-415 water bomber [Viking, 2021], the goal for an autonomous ensemble is to start countermeasures as soon as possible [Roldán-Gómez et al., 2021]. Approaches for equipping UAV accordingly either adapt the principle of manned water bombers [Qin et al., 2016], unfortunately, limited in their load capacity when used individually (the Boeing cargo drone, e.g., can carry up to 227 kilograms [Boeing, 2019]), or use innovative techniques like extinguishing balls [Aydin et al., 2019] that release chemicals to extinguish a fire when exposed to high temperatures.

Possible SCORe Mission for Flying Ensembles

For illustrating the feasibility of handling forest fires with our approach to Mission Programming for Flying Ensembles by Combining Planning with Self-Organization, we use the following simplified scenario of a forest fire. We assume a firefighter handling a situation where the occurrence of spontaneous forest fires is very likely (cf. Figure 2.8). Thus, on a large scale, forest fires may ignite spontaneously in a specific area. The firefighter's equipment consists of a flying ensemble, i.e., multiple UAVs, and a set of additional hardware modules, i.e., S&A, that can be attached to each of the UAVs. Each UAV is pre-configured with a set of S&A. That configuration can originate, e.g., from the last SCORe mission the ensemble was used in or because of initial investigations the firefighter performed beforehand concerning the most likely situation the ensemble will face (cf. Figure 2.8a). The firefighter then needs to appropriately instruct the ensemble with a SCORe mission consisting of the three phases:

- Search the whole forest for finding the critical area, e.g., by measuring humidity and temperature and evaluating measurements on the likelihood of a spontaneous fire ignition (cf. Figure 2.8b)
- moving the whole ensemble to this area,
- Continuously Observe the area for detecting fires (cf. Figure 2.8c),
- React to detected fires by trying to extinguish them as fast as possible (cf. Figure 2.8d).

Because of the potential sizeable spatial expansion of the forest, the size of the ensemble, and other urgent tasks only a human can accomplish, it is not always a feasible option for the firefighter to manually control all UAVs, define routes for all UAVs in the flying ensemble, or to react ad-hoc to newly identified fires. Instead, in such a situation, the firefighter more likely



(a) firefighter with ensemble and S&A

(b) Search for the critical area

(c) Continuously Observe the area

(d) React to detected fires

Figure 2.8: Firefighters orchestrating an ensemble to deal with a forest fire scenario.

needs to specify how the ensemble should react in different situations and let the ensemble act appropriately and as autonomously as possible. Thus, our approach for Mission Programming for Flying Ensembles by Combining Planning with Self-Organization can improve the firefighters' situation, as demonstrate in the following chapters.

Chapter Summary and Outlook

In this chapter, we introduced the problem domain of Search, Continuously Observe, and React (SCORe) missions subsuming other problem domains currently found in literature, namely Environmental Monitoring, Distributed Surveillance, Search and Rescue, and Major Catastrophe Handling. We further analyzed current approaches from related work and categorized them according to our definition of SCORe. Additionally, we introduced different case studies from the domain of Environmental Monitoring (i.e., using flying ensembles for collecting spatially distributed data used to improve climate models and for testing novel measurement methods) and Major Catastrophe Handling (i.e., using flying ensembles for helping rescue forces to deal with gas accident and forest fires) which we use for demonstrating the benefits of our approach of Combining Planning with Self-Organization for Mission Programming for Flying Ensembles and evaluating them respectively.

We now propose the general idea for how we enable the same flying ensemble to be applied to these different use cases by introducing the reference software architecture building the base for our approach.

Chapter 3

The Idea of Multipotent Systems: Separating Capabilities from Agents

Summary. In this chapter, we describe our understanding of Multipotent Systems, which we introduce as a new type of classification type for Multi-Agent Systems (MAS)/Multi-Robot Systems (MRS) besides the currently existing types of homogeneous and heterogeneous systems. In Multipotent Systems, we leave behind the idea of having agents and capabilities connected permanently after design-time and instead propose to let them be flexibly configurable at run-time. We elaborate on the benefits arising from this new perspective, derived from combining the advantages of homogeneous systems with that of heterogeneous systems while avoiding their current drawbacks: Within Multipotent Systems, we subsume robustness and scalability derived from homogeneous systems and flexibility derived from heterogeneous systems. We find that Multipotent Systems require new and adapted approaches for algorithms and technologies currently applied to MAS/MRS. We introduce a layered reference architecture for such Multipotent Systems and describe how we can use it for designing SCORe missions, deriving plans from that missions, work through that plans with autonomously formed flying ensembles, and realize the possibility for the run-time adaptation of the configuration of agents concerning their hardware and capabilities. We guide through that chapter by applying the described concepts and their interwoven interplay to a running example motivated from by the case study of *Improving Climate Models*. Further, we describe our prototypical implementation of this layered reference architecture for Multipotent Systems, which we realize with the multi-agent framework Jadex Active Components Framework (Jadex) [Braubach and Pokahr, 2012]. We conclude this chapter by evaluating the feasibility of deploying this prototypical implementation to the real world, involving multiple field and laboratory experiments motivated by the case studies Improving Climate Models, and the case study Innovative Measurement Methods performed with real hardware.

Publication. Contents of this chapter have been published in [Kosak, 2017; Kosak et al., 2018; Kosak, 2018; Kosak et al., 2019; Wanninger et al., 2018].

3.1 Introducing Multipotent Systems and Their Benefits

The last two decades have seen a multitude of different applications and frameworks all having the purpose to deploy ensembles for versatile use cases, which we highlighted in Section 1.2. The bandwidth ranges from SAR [Murphy et al., 2008; Daniel et al., 2009; Scherer et al., 2015; De Cubber et al., 2013] over such used for Distributed Surveillance [Martinez et al., 2014; Matikainen et al., 2016; Zhang et al., 2012] to those dedicated to Environmental Monitoring [Wolf et al., 2017; Duarte et al., 2016; Thenius et al., 2016], among others. The sheer number of different applications, robot designs, and programming frameworks shows: For each new use case, the wheel is reinvented over and over again, new robots with new capabilities for new applications have to be invented, designed, built, and programmed with individualized software. We see, requirements concerning the sensing and acting capabilities of agents (e.g., measuring certain parameters or interacting with the environment) typically change when switching from one use case to another. Like we illustrate in Chapter 2, in each different case study and further in every phase of a respective SCORe mission, agents acting in ensembles may require different capabilities for working on them. For fulfilling these requirements, current approaches introduce appropriately configured robots (concerning hardware and software) for each different purposes [Scherer et al., 2015; De Cubber et al., 2013; Martinez et al., 2014; Duarte et al., 2016]. Traditionally, these systems are either classified to be homogeneous or heterogeneous concerning the hardware configuration of robots in that systems [Stone and Veloso, 2000] (cf. our clarifying definitions in Section 1.2.1). Different hardware configurations determine the set of available capabilities the agents controlling the robots can execute. Thus, while agents in homogeneous systems all have the same capabilities, in heterogeneous systems they typically have different capabilities. This system property poses different advantages and drawbacks we classify in the following.

3.1.1 Classifying the System Type of Homogeneous Systems

Homogeneous systems can compensate for failures of individual agents because all other agents in the system have the same capabilities available and thus can take over the task of the failing agent in principle, i.e., homogeneous systems *can provide high robustness* [Brambilla et al., 2013]. Further, because homogeneous systems often can function without a central controlling instance and rely on neighbor communication instead, the number of agents in homogeneous systems can be very high, i.e., homogeneous systems *can provide high scalability* [Barca and Sekercioglu, 2013]. This enables the application of homogeneous systems also in spatially wide areas, e.g. when adapting swarm behavior in swarm algorithms applied to robots [Brambilla et al., 2013]. But homogeneous systems also face drawbacks reducing their general applicability [Nedjah and Junior, 2019]. As all agents provide the same capabilities, the bandwidth of specialization for different purposes within the same system is limited. This limitation becomes obvious in applications from the field of Swarm-Robotic Systems where the range of capabilities for agents is limited in the number of S&A a agent can operate simultaneously due to physical properties. A homogeneous system thus is often limited to the one specific tasks it is designed for, i.e., homogeneous systems *often lack flexibility* [Nedjah and Junior, 2019]. Caused by this lack of flexibility, homogeneous systems often cannot exploit the results derived when accomplishing the task they are designed for, i.e., homogeneous systems *are not fit for handling complex missions* like SCORe missions. Approaches for achieving such require the application of multiple homogeneous systems working together instead, e.g., like

proposed by Dorigo et al. [2013].

3.1.2 Classifying the System Type of Heterogeneous Systems

Heterogeneous systems instead offer other advantages and suffer from other drawbacks, being complementary to that of homogeneous systems. In heterogeneous systems, compensating for failures in individual agents requires high effort because there not necessarily exists another agent that provides the same capabilities as the failing agent. Thus, compensating for failures without required redundancy can often only be achieved by complex recovering strategies, e.g., generating new plans for the system that work without the failing instance [Coltin and Veloso, 2013]. Thus, heterogeneous systems *often lack sufficient robustness* [Brooks et al., 2016]. When scaling the application size or the spatial area where a heterogeneous system should be applied, another drawback is revealed: Heterogeneous systems then require complex planning, scheduling, and coordination mechanisms for dealing with the situation, sometimes exceeding the range of computability within appropriate time [Hudziak et al., 2015]. Thus, heterogeneous systems *can be limited in their scalability.* But heterogeneous systems also provide one big advantage: Because of the diversity in hardware configurations of robots, agents in heterogeneous systems can provide a very broad bandwidth of different capabilities [Rizk et al., 2019]. The problem that robots can only carry a certain maximum weight does not hinder flexibility for tasks like it does in homogeneous systems: If a certain heavy hardware module needs to be carried by a robot for enabling its agent to provide a respective capability, in heterogeneous systems, we can include a specialized robot with the required properties into the system. Hood et al. [2017], e.g., support UAVs by using UGVs providing computational power with a hardware module that is too heavy to be carried in the air. Thus, heterogeneous systems *can achieve goals even in complex missions* that consist of different tasks that can also require very specific capabilities each. Thereby, heterogeneous systems *can provide a very high degree of flexibility.*

3.1.3 Drawbacks when Focusing on Homogeneous or Heterogeneous Only

Current consequences are that system architects need to decide on one of these two options, i.e, whether a homogeneous or heterogeneous system fits better to their respective use-case and application. Thereby, they also have to decide on how to handle the tradeoff between the aforementioned system properties *robustness*, *scalability*, and *flexibility* none of the existing system types can provide at the same time. Further, this approach leads to massive resource and engineering overhead: The amount of needed robots grows proportionally with the number of different applications, use cases, and tasks.

3.1.4 Multipotent Systems to Avoid Drawbacks

To address these drawbacks, we propose to separate the concept of capabilities from that of agents for overcoming the current state of looking at MAS/MRS. Viewed from this new perspective, a robot is a conglomerate of multiple hardware modules (i.e., S&A) attached to a platform hosting the actual agent who can autonomously make use of attached S&A for deriving capabilities and accomplishing tasks by executing these capabilities. By assuming hardware modules to be flexibly reconfigurable at run-time, we thus enable the ad-hoc, *physical reconfiguration* of robots. Thus, agents controlling the robots can dynamically gain and lose

capabilities which we can steer to happen as required by the respective application and use-case at run-time.

Definition: Physical Reconfiguration A physical reconfiguration of a robot is a modification, applied to the existing hardware configuration of one (or multiple) robots which due to the modified configuration(s) can result in a changed set of capabilities for the agent controlling the robot.

Many of those agents working together then form an ensemble of reconfigurable agents. We name this new hybrid form of a system class, where agents' capabilities can be reconfigured by changing the physical composition of the robots they control, a *Multipotent System*:

Definition: Multipotent System The term *multipotency* is an analogy from biology. According to Giorgetti et al. [2012], multipotent progenitor cells can adapt their functionality when brought into new environments.

We transfer that property for agents in ensembles that can adapt their set of provided capabilities when necessary, i.e., when requirements defined by SCORe missions change during run-time. Multipotent Systems are characterized by their homogeneity concerning the software agents (homogeneity at design time) and the possibility for becoming heterogeneous concerning their provided capabilities when needed (heterogeneity at run-time).

By this, Multipotent Systems inherit benefits from both kinds of current system types at the same time as they can omit their drawbacks. On the one hand, with their *run-time heterogeneity*, agents of Multipotent Systems can be configured very individually to reproduce the versatility of heterogeneous systems. On the other hand, with their *design-time homogeneity*, agents in Multipotent Systems can provide robustness in the case of failures while the size of applications can scale high when adapting respective behavior, e.g., swarm algorithms.

For achieving this, we need to develop new technologies and integrate them with already existing ones: In addition to equipping the system with *Planning* and *SO* abilities, we enable ensembles implementing the layered reference architecture for Multipotent Systems we propose in the following to be adapted *before run-time* and autonomously (self-) adapt to changing requirements *at run-time*, following the paradigm of organic computing [Müller-Schloer et al., 2011]. In this chapter, we focus on the software architecture necessary for integrating these approaches. We thereby also tackle the challenges in designing software for robots swarms that, according to Barca and Sekercioglu [2013]

> "include the following: (1) Selecting appropriate centralized or decentralized communication and control schemes. (2) Incorporating important behaviors and traits such as self-organization, scalability, and robustness. (3) Devising mechanisms that support goal-directed formations, control, and connectivity. (4) Implementing mapping, localization, path planning, obstacle avoidance, object transport, and object manipulation functions that enable swarms of robots to interact efficiently with the environment. (5) Addressing problems related to energy consumption."

In Section 3.2, we illustrate how our concept of a layered reference architecture for Multipotent Systems supports achieving this.

3.2 Layered Reference Architecture for Multipotent Systems

To realize our idea of Multipotent System combining the strengths of homogeneous and heterogeneous systems while avoiding their weaknesses, we elaborated on in Section 3.1, we propose a layered software architecture as a reference. This architecture consisting of the four layers PLAN LAYER, ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER serves as a platform to support all necessary functionality. Each layer encapsulates its concepts and algorithms designed to enable our approach of Combining Planning with Self-Organization aiming at the problem of Mission Programming for Flying Ensembles. In the following Sections 3.2.5 to 3.2.8, we introduce this architecture, including a specification of purposes for each of its layers. We describe how needed technologies and algorithms are situated within and distributed over the different layers of the architecture and how they integrate. We support these explanations with a running example we introduce in Section 3.2.4. In the following chapters Chapters 4 to 7, we then deliver the detailed explanations for algorithms and mechanisms with complex examples. That way, we can focus on the interplay of all required technologies first without unnecessarily increasing complexity.

3.2.1 Assumptions

We make the following assumptions necessary for the seamless interaction of our concepts:

- **Communication Medium is Available:** We assume reliable communication between agents for all algorithms, i.e., assume that sent messages eventually arrive. We can assure such, e.g., with WiFi meshes like they are used by other approaches focusing on flying ensembles [Pojda et al., 2011; Scherer et al., 2015; Yuan et al., 2018; Stellin et al., 2020].
- **Energy Costs for Communication and Computations are Negligible:** Further, when using UAV in flying ensembles as we intend to, the most energy is consumed by the motors and rotors used for realizing the actual flying movement. Thus, we assume the energy cost required for communication and calculations to be that low that we can neglect it during design, which stands in contrast to the design challenges defined by Barca and Sekercioglu [2013] we listed in Section 3.1. Instead, we consider the possibility of total energy depletion in accumulators of UAV as a case of agent failure and handle it the same way.
- **Agent Failures:** We assume that we can compensate for agent failures with appropriate SO-mechanisms, i.e., by exploiting the robustness of swarm algorithms. While we give a conceptual idea on how we can provide robustness also outside of SO-mechanisms and give first preliminary results of a prototypical implementation in Section 7.7, we do not yet include such mechanisms in the descriptions concerning our reference architecture and assume that failures do not occur in these situations.
- **Self-Descriptive Hardware is Available:** To bridge the gap to reality, we further assume modular and reconfigurable hardware which agents can access over respective capability interfaces. We achieve this by using Self-Descriptive Hardware (SDH) offering semantic annotations to agents describing their provided capabilities. Over a unified interface, software agents can read these descriptions, become self-aware of their available

capabilities, and know how to execute these capabilities. While we elaborate on methods and techniques for realizing that in [Eymüller et al., 2018; Wanninger et al., 2018; Kosak et al., 2018; Wanninger et al., 2021] originating from the same research project COMBO[1], we do not further investigate the technical details in this thesis. Instead, we only investigate the software interface and the required concepts for accessing Self-Descriptive Hardware (SDH) and give a superficial description of their physical hardware interface only. Thus, we assume to have a working resources management when using SDH, meaning we do not need to schedule hardware access and rely on the hardware implementation to care for that issue. Further details concerning self-descriptive hardware then are investigated in the doctoral thesis of Constantin Wanninger, focusing on *Semantic Plug and Play - Fähigkeitsbasierte Hardwareanbindung modularer Robotersysteme*[2].

3.2.2 Definitions and Notation

We briefly introduce the notation necessary for describing the interplay of different technologies within our reference architecture. Combined with the glossary, this section can be used as a reference guide. In the following, we depict agents within the Multipotent System as $\alpha \in \mathcal{A}_{\mathcal{MS}}$. The S&A that we enrich with relevant data we can use for letting them describe themselves in a machine-interpretable way (encapsulated in a so-called Semantic Shell [Wanninger et al., 2021]), we call Self-Descriptive Hardware $\text{SDH} \in \mathcal{SDH}_{\mathcal{MS}}$. An agent α is configured with a set $\mathcal{SDH}_\alpha$ consisting of $\text{SDH} \in \mathcal{SDH}_{\mathcal{MS}}$. We can further specify this configuration for a specific state of the respective $\mathcal{MS}$, i.e., a configuration $\mathcal{SDH}_\alpha^\rho$ specifies the configuration of agent α when a plan ρ is active. This set $\mathcal{SDH}_\alpha$ then defines the set of capabilities $\mathcal{C}_\alpha$ consisting of the single capabilities $c \in \mathcal{C}_\alpha$ the agent α can provide. Capabilities $c \in \mathcal{C}$ can be of two types. Either they are physical capabilities $c_\Upsilon^p \in \mathcal{C}^p$ directly provided by an SDH, or they are virtual capabilities $c_\Upsilon^v \in \mathcal{C}^v$ provided by the combination $\bigstar$ of multiple c_Υ^p from $\mathcal{SDH}_\alpha$, where Υ depicts the concrete type of the respective capability.

Because weight is relevant for movement in flying ensembles, each SDH offers information on its weight in the self-description of its Semantic Shell. If the weight of an SDH is negative, this SDH enables the respective agent it is configured to with an additional payload. That way, we can define a weight-function ω to describe the total weight of an agent's configuration, i.e., applying $\omega(\mathcal{SDH}_\alpha)$ results in a total weight indicating whether the agent can still move (total weight ≤ 0) or not (total weight > 0).

Together, $\mathcal{A}_{\mathcal{MS}}$ and $\mathcal{SDH}_{\mathcal{MS}}$ define the Multipotent System $\mathcal{MS}$, i.e., $\mathcal{MS}{:=}$ ($\mathcal{A}_{\mathcal{MS}}$, $\mathcal{SDH}_{\mathcal{MS}}$). A configuration $\zeta(\mathcal{SDH}_{\mathcal{MS}})$ expresses a set-partitioning of all $\text{SDH} \in \mathcal{SDH}_{\mathcal{MS}}$, describing the distribution of hardware to the different agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ defining the set of $\mathcal{SDH}_\alpha$ for every agent.

SCORe missions for that $\mathcal{MS}$ are formulated as a Hierarchical Task Networks (HTN) [Georgievski and Aiello, 2015] dedicated to the use case they handle. A Hierarchical Task Networks (HTN) designed for the specific SCORe mission MIS thus is denoted as HTN_{MIS}. A HTN consists of multiple partial plans PART-ρ the designer of the HTN can interconnect in a network according to its needs. Plans ρ are generated composing the partial plans from HTNs

[1] COMBO - *Kombination von Planung, Selbst-Organisation und Rekonfiguration in einem Roboterensemble zur Ausführung von SCORe Missionen* - grant number 402956354.

[2] German title, translated by the author: Semantic Plug & Play — Capability-Based Hardware Integration of Modular Robot Systems

by using automated planning taking into account the current environmental conditions and the internal state of $\mathcal{MS}$, subsumed in a world state WS.

Ensembles are sets of agents formed to handle specific plans. An ensemble $\mathcal{E}^\rho$ formed for handling a specific plan ρ derived from a $\text{HTN}_{\text{MISSION}}$, thus is a subsystems of the original $\mathcal{MS}$, i.e, $\mathcal{E}^\rho := (\mathcal{A}_{\mathcal{E}^\rho}, \mathcal{SDH}^{\mathcal{E}_\rho})$ with $\mathcal{A}_{\mathcal{E}^\rho} \subseteq \mathcal{A}_{\mathcal{MS}}$ and $\mathcal{SDH}^{\mathcal{E}_\rho} \subseteq \mathcal{SDH}_{\mathcal{MS}}$. Nevertheless if we write $\alpha \in \mathcal{E}^\rho$ in the following, we simply address one of the agents from $\mathcal{A}_{\mathcal{E}^\rho}$. Each plan ρ derived from a HTN consists of execution information encoded in a set of tasks $t \in \mathcal{T}^\rho$ for PLAN WORKERs and coordination information $\mathcal{CI}^\rho$ for a PLAN COORDINATOR, i.e., $\rho := (\mathcal{CI}^\rho, \mathcal{C}_t)$. Each $t \in \mathcal{T}^\rho$ contains the necessary information on requirements concerning capabilities $c \in \mathcal{C}_t$ for assigning the task and an associated cooperation patterns $\mathcal{CP}_t^\rho$ defining when and how agents $\alpha \in \mathcal{A}_{\mathcal{E}^\rho}$ interact within the ensemble $\mathcal{E}^\rho$. In a state of the $\mathcal{MS}$ where an ensemble $\mathcal{E}^\rho$ for ρ can be formed, the configuration can be described as $\zeta(\mathcal{SDH}^\rho_{\mathcal{MS}})$. In such a configuration, there is a set of agents forming the respective ensemble, that as a collective, offers all requirements defined by the plan, i.e., there is an injective function mapping one agent all tasks $t \in \mathcal{T}^\rho$ while the respective agent provides all required capabilities in that task.

3.2.3 General Design Decisions and Overview on the Architecture

We designed the Multipotent System reference architecture with high flexibility concerning its application in different use cases in mind. To achieve this flexibility, we conceptually separate capabilities from the agents executing them and make robots physically reconfigurable. This design decision has severe consequences for the general design of the architecture and the technologies we decided to integrate. The Multipotent System reference architecture consists of the four layers PLAN LAYER, ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER that each agent in the system needs to implement, resulting in a homogenous system design. We give a graphical overview of our architecture in Figure 3.1. For achieving flexibility during the execution of a SCORe mission, the Multipotent System provides necessary self-awareness and self-organization features, where each layer is responsible for contributing its very own part. In general, to obtain modularity, communication is restricted to only happening between vertically adjacent layers (internally, i.e., on the same agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$) and horizontally adjacent layers (externally, i.e., between two or more agents $\alpha_{i \neq j} \in \mathcal{A}_{\mathcal{MS}}$). We allow for two exceptions from that scheme:

1. Communication between ENSEMBLE LAYER and AGENT LAYER is also necessary between two or more different agents $\alpha_{i \neq j} \in \mathcal{A}_{\mathcal{MS}}$ for enabling the coordinated execution of SCORe missions, e.g., we allow the ENSEMBLE LAYER on agent α_1 to communicate *diagonally* with the AGENT LAYER of other agents $\{\alpha_{2,\ldots,n}\}$ if necessary.
2. External communication between the user of the Multipotent System and the agents in $\mathcal{A}_{\mathcal{MS}}$ is necessary on every layer for allowing the user to supervise the Multipotent System and intervene in its autonomy if necessary, e.g., for adapting the configuration $\zeta(\mathcal{SDH}_{\mathcal{MS}})$ of a Multipotent System.

Besides enabling modularity, with this design, we can combine the situation-aware, ***automated planning*** of SCORe missions, performed by the interplay of the PLAN LAYER and the ENSEMBLE LAYER, with the ***SO execution*** of generated plans, achieved by appropriate SO-mechanisms handling the necessary coordination between ENSEMBLE LAYER and AGENT LAYER. By further combining those techniques with the concept of *semantic plug & play* realized on SEMANTIC HARDWARE LAYER integrating self-descriptive hardware (SDH) in a standardized way,

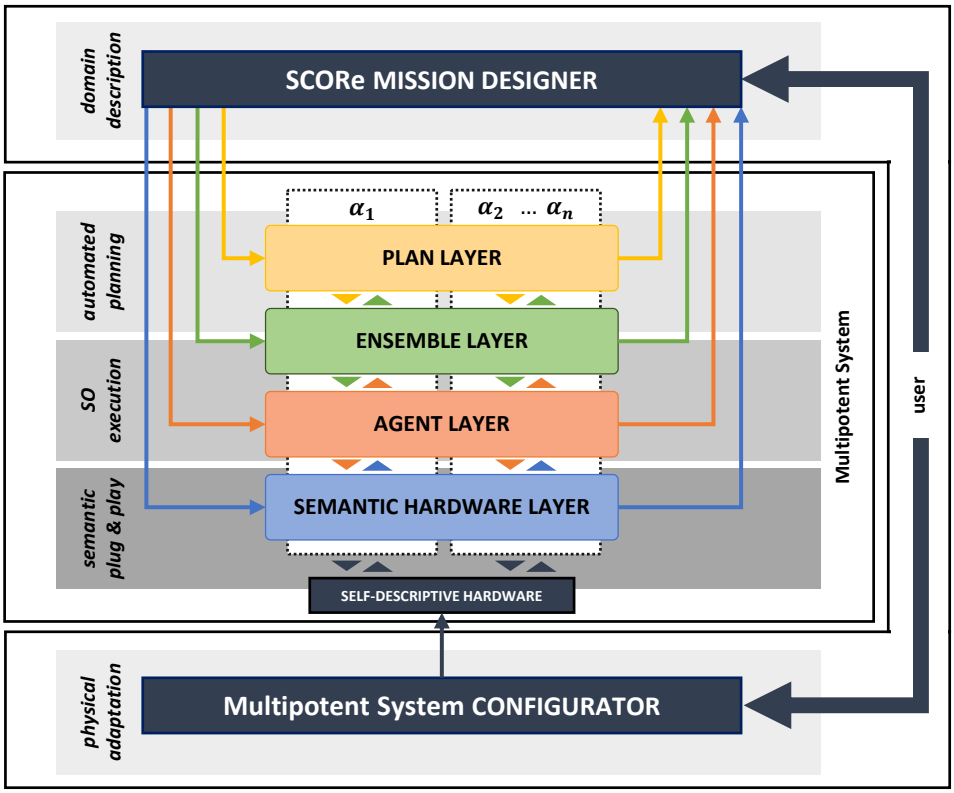


Figure 3.1: Overview of the layered Multipotent System reference architecture. Each of the n agents in the Multipotent System $\mathcal{MS} := (\mathcal{A}_{\mathcal{MS}}, \mathcal{SDH}_{\mathcal{MS}})$ is designed homogeneously to each other agent implementing the depicted layers PLAN LAYER, ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER. To abstract from concrete S&A and their implementation, we assume to have self-descriptive hardware SDH $\in \mathcal{SDH}_{\mathcal{MS}}$ available. SDH provide a uniform software interface offering information on their available capabilities accessible by the SEMANTIC HARDWARE LAYER while also allowing agents for executing capabilities when required. A *semantic plug & play* mechanism allows for the flexible physical adaptation of robots influencing the respective agents' available capabilities. *Physical adaptation*, i.e., modifications of $\zeta(\mathcal{SDH}_{\mathcal{MS}})$, can be performed by the system's user in its role of a Multipotent System CONFIGURATOR. By appropriately restricting the communication between layers (arrows), we can take care of the modularity we require for this flexibility on the one hand. On the other hand, this design also allows for the *SO execution* of situation-specific plans generated by *automated planning* derived from SCORe missions. The *domain description* relevant for a SCORe mission can be created by the system's user in its role of a SCORe MISSION DESIGNER beforehand or at run-time.

the Multipotent System can flexibly adapt to the requirements the system's user defines during performing the *domain description* including the *SCORe mission design*.

Figures 3.1 to 3.3 serve as overview of all concepts necessary for realizing the Multipotent System reference architecture. While we use the following Sections 3.2.5 to 3.2.8 to describe all these concepts in detail, we briefly give an overview here first. In Figure 3.2, we illustrate the prototypical activity of handling a SCORe mission with a Multipotent System. We depict

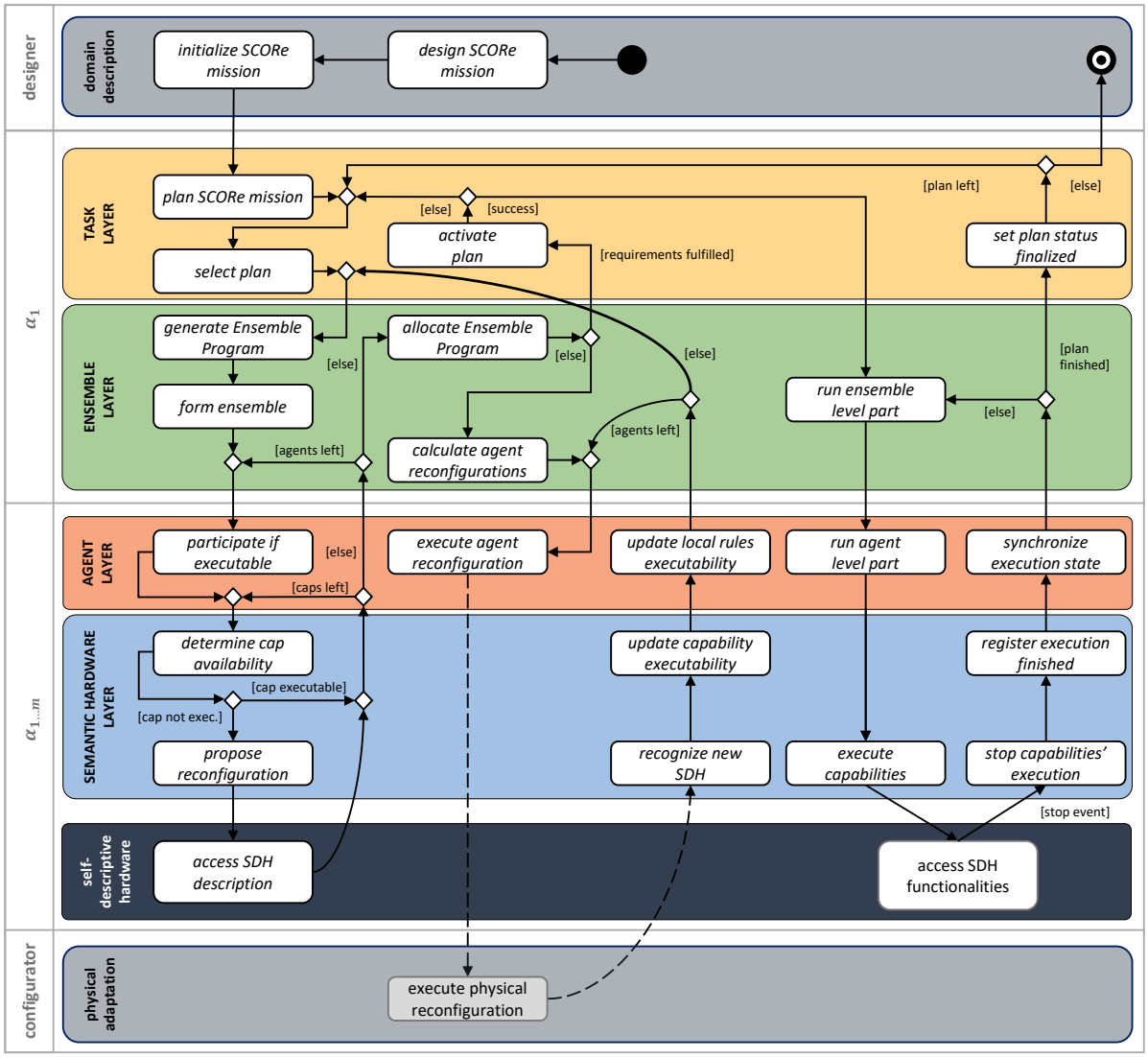


Figure 3.2: Exemplary control flow describing the execution of a SCORe mission performed by the agents $\alpha_{1,\ldots,n} \in \mathcal{A}_{\mathcal{MS}}$ as activity diagram. While the activities on PLAN LAYER and ENSEMBLE LAYER are executed by one coordinating agent α_1, the activities on the AGENT LAYER and SEMANTIC HARDWARE LAYER involve all agents of the respective ensemble. The interaction with the system's user in the role of a SCORe MISSION DESIGNER is only necessary for the initializing the execution for and performing adaptations to $\zeta(\mathcal{SDH}_{\mathcal{MS}})$ in the role of a CONFIGURATOR if this is required during the execution of a SCORe mission.

the necessary interactions within $\mathcal{MS}$ consisting of agents $\alpha_{1,\ldots,n} \in \mathcal{A}_{\mathcal{MS}}$, including their internal interactions (bold arrows) and external interactions with the user (dashed arrows). For implementing these activities and depending on their function during the handling of a SCORe mission, agents $\alpha_{1,\ldots,n} \in \mathcal{A}_{\mathcal{MS}}$ need to adopt different roles on the different layers depicted in Figure 3.3. Besides, also the user needs to adopt different roles and interact with the ensemble during this process: On the one hand, the user acts as an initiator for letting

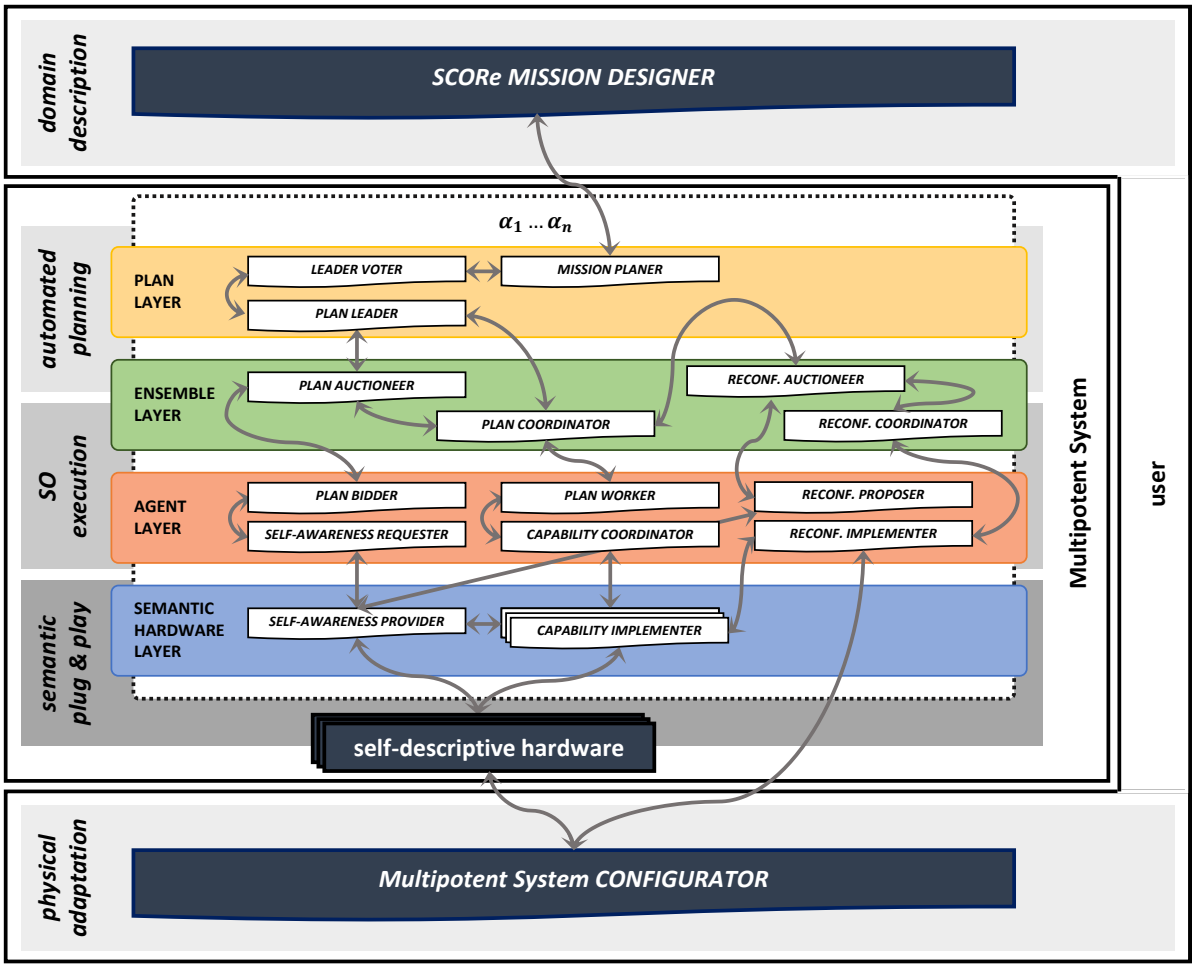


Figure 3.3: The different roles each agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ can adopt on the PLAN LAYER, ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER when implementing the Multipotent System reference architecture. We depict the communication and data exchange between different roles the agents can adopt in an ensemble formed for handling a SCORe mission including the necessary interaction with SDH $\in \mathcal{SDH}_{\mathcal{MS}}$ with arrows. Further, we illustrate how the user of the Multipotent System can interact with the Multipotent System for initializing the handling of a SCORe mission for performing physical adaptations concerning the set $\mathcal{SDH}_{\alpha}$ of every agent $\alpha \in \mathcal{MS}$ during the execution of a SCORe mission.

the ensemble work on a SCORe mission by designing and introducing the mission. On the other hand, the user acts as a configurator of the Multipotent System if the ensemble requires physical reconfigurations during the execution of a SCORe mission.

We now describe the interplay between the different roles we depict in Figure 3.3 on the different layers of our architecture we depict in Figure 3.1 in different mechanisms and algorithms necessary for realizing the different activities we illustrate in Figure 3.2.

Table 3.1: Capability definitions for the running example.

name	description	parameter type	return type
$c^p_{\text{MV-POS}}$	moving to a geographic coordinate	⟨ LAT, LON, ALT ⟩	VOID
$c^p_{\text{MV-VEL}}$	moving with a velocity / trajectory	⟨ X,Y,Z ⟩	VOID
$c^p_{\text{M-POS}}$	own position in geographic coordinates	VOID	⟨ LAT, LON, ALT ⟩
$c^p_{\text{M-VEL}}$	own velocity as vector in meters per second	VOID	⟨ X,Y,Z ⟩
$c^p_{\text{M-PM}}$	concentration of PM in ppm	VOID	DOUBLE
$c^p_{\text{M-TEMP}}$	temperature in degree of Celsius	VOID	DOUBLE
$c^p_{\text{M-HUM}}$	relative humidity in percent of max saturation	VOID	DOUBLE
$c^p_{\text{M-WIND}}$	wind direction as 3-dimensional vector	VOID	⟨ X,Y,Z ⟩

3.2.4 Running Example: Determining the Particulate Matter Concentration by Analyzing the Nocturnal Boundary Layer

We illustrate our descriptions in this Sections 3.2.5 to 3.2.8 with a running example from the case study of *Improving Climate Models* we introduced in Section 2.3. Every time we continue this example, we shade the respective text with a grey color. Because we focus on the interplay of the different technologies in this chapter, we keep the running example as simple as possible and provide complex examples when investigating the different technologies and mechanism isolated in the respective Chapters 4 to 7.

Running Example For the sake of simplicity, we use a reduced scenario situated in the case study of *Improving Climate Models*: As a user of a Multipotent System, we want to estimate the current concentration of Particulate Matter (PM) and its potential impact on the health of residents. We therefore specify a SCORe mission requiring a flying ensemble to

1. determine the occurrence and location of a temperature inversion in the Nocturnal Boundary Layer (NBL) with a flying ensemble in a relevant area r, with a height of h (cf. Figures 2.2 and 2.3), i.e, the Search phase of the SCORe mission,
   a) if existent, let an ensemble monitor the current weather locally for determining unfavorable conditions, then continue with 2.
   b) otherwise, return the ensemble to the Multipotent System's user

2. monitor weather conditions until one of the following events happen (Continuously Observe phase of the SCORe mission):
   a) if weather conditions get critical (i.e., wind stream to inhabited area) let the ensemble determine the current concentration of PM, then continue with 3.
   b) if the user aborts the observation, return the ensemble to the user
3. determine the criticality of the current PM concentration *con* and decide (React phase of the SCORe mission):
   a) If the PM concentration gets critical, distribute this information to interested parties and return to monitoring the weather conditions in 2.
   b) otherwise return to monitoring the weather conditions in 2.

The concrete Multipotent System $\mathcal{MS}_{\text{NBL}} = (\mathcal{A}_{\text{NBL}}, \mathcal{SDH}_{\text{NBL}})$ we have at hand in this reduced scenario consists of three agents $\mathcal{A}_{\text{NBL}} = \{a_1, a_2, a_3\}$ and a set of five Self-Descriptive Hardware $\mathcal{SDH}_{\text{NBL}} = \{\text{SDH}_1, \text{SDH}_2, \text{SDH}_3, \text{SDH}_4, \text{SDH}_5\}$ providing the capabilities required for solving the SCORe mission in the example. Thus, we ascribe capabilities in this scenario used for *moving (i.e., flying) in three dimensional space* as $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-VEL}}$, having a specific input but no concrete return value (VOID) each. Capabilities for *measuring different parameters* of interest do not have concrete input values but return their respective measurements encoded with appropriate data types. We depict these capability definitions in Table 3.1

We thus further can describe $\mathcal{SDH}_{\text{NBL}}$ used in the running example by defining the capabilities they provide to the agent they are associated with in Table 3.2. A negative weight indicates that when configured with that SDH, an agent can carry an additional payload of the respective weight. Because we focus on mobile and especially flying ensembles, besides information on capabilities provided by the SDH, the respective weight of each SDH is relevant. Exceeding the maximum payload of an agent, i.e., the total weight that can be carried might impact the possibility of executing specific capabilities. Especially capabilities for moving an agent in space can become unavailable if the total weight gets too high. Thus we also have to respect the total weight of an agent when it is equipped with SDH if we want all capabilities provided by the set of SDH to be available to the agent. Because we denote actuators enabling the movement of an agent with negative weight to indicate that the agent can carry additional SDH, we can sum up all weights of an agent to verify whether capabilities for movement are still available.

3.2.5 The Plan Layer

The PLAN LAYER serves as a relay between the Multipotent System's user and the Multipotent System's agents $\mathcal{A}_{\mathcal{MS}}$. For easing the design of SCORe missions (cf. *design score mission* Figure 3.2) in the role of a SCORE MISSION DESIGNER (cf. Figure 3.3) and ensureing a user-friendly interface between the system and its user, we propose to use a domain-specific problem description language appropriate for many use-cases and applications from the class of SCORe missions.

Table 3.2: Hardware modules SDH $\in \mathcal{SDH}_{\text{NBL}}$ used in the running example.

SDH $\in \mathcal{SDH}_{\text{NBL}}$	description	capabilities	WEIGHT
SDH_1	precise PM sensor	$\{c^p_{\text{M-PM}}\}$	$+5kg$
SDH_2	UAV	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$	$-5kg$
SDH_3	UAV with integrated temperature sensor	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-TEMP}}\}$	$+1kg$
SDH_4	combined weather sensor	$\{c^p_{\text{M-WIND}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-HUM}}\}$	$+5kg$
SDH_5	UAV	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$	$-5kg$

3.2.5.1 Defining SCORe Missions

We therefore rely on the concepts of Hierarchical Task Networks (HTN) as introduced by Erol et al. [1994] (cf. the survey on HTN in [Georgievski and Aiello, 2014]). HTNs are especially handy for autonomous systems deployed in real-world environments and enable the design of reusable, partial plans PART-ρ. We can compose and instantiate such partial plans with relevant parameters for different SCORe missions in a modular fashion [Magnenat et al., 2009]. With their graph-like structure HTNs further can be easily encoded in a machine-readable way, simplifying the activity *initialize SCORe mission* for the user (cf. Figure 3.2). In the way we propose to use them, HTNs offer a promising combination of (1) external user control and (2) an appropriate degree of freedom and robustness, enabling the executing Multipotent System for autonomous decisions. On the one hand, we can achieve (1) by using the HTNs' inherent concept of problem decomposition where experts of the problem domain can define within a Complex-Node (CN) how complex and abstract instructions can be decomposed into partial plans. Partial plans then can consist of other Complex-Nodes, one Primitive-Node (PN), or multiple PNs holding concrete instructions for the system. On the other hand, we can achieve (2) by letting an ensemble execute instructions from PNs with SO-mechanisms. The SO-mechanism-based execution of PNs allows the executing ensemble for responding to and compensating for unforeseen (i.e., not planned) situations at run-time. This way, we face the fact that by default, approaches for automated planning using HTNs have only minor uncertainty handling strategies, which can result in extensive and time-consuming planning [Koenig, 2001]. Integrating automated planning using HTN (cf. *plan SCORe mission* in Figure 3.2) with the paradigm of SO thus aims at reducing the frequency of planning and reducing the necessary level of detail during planning. We can achieve this by introducing PNs on the collective level. Instructions on the collective level do not need to be further decomposed for individual agents in the ensemble (which is the strategy of many current approaches for MAS/MRS, e.g., that in [Nissim et al., 2010; Obst and Boedecker, 2006; Brenner and Nebel, 2009; Elkawkagy and Biundo, 2011; Ed Durfree, 2013]) as they define the necessary local interaction inherently. In situations where the result of an agent's action does not comply with the predicted effect of a planned PN, classic planning approaches need to calculate new plans accounting for this discrepancy. Such situations can occur quickly, e.g., if an agent executing a search task breaks down, the object of interest is not found, and the actual state differs from the assumptions on the world state made during planning time. In our approach,

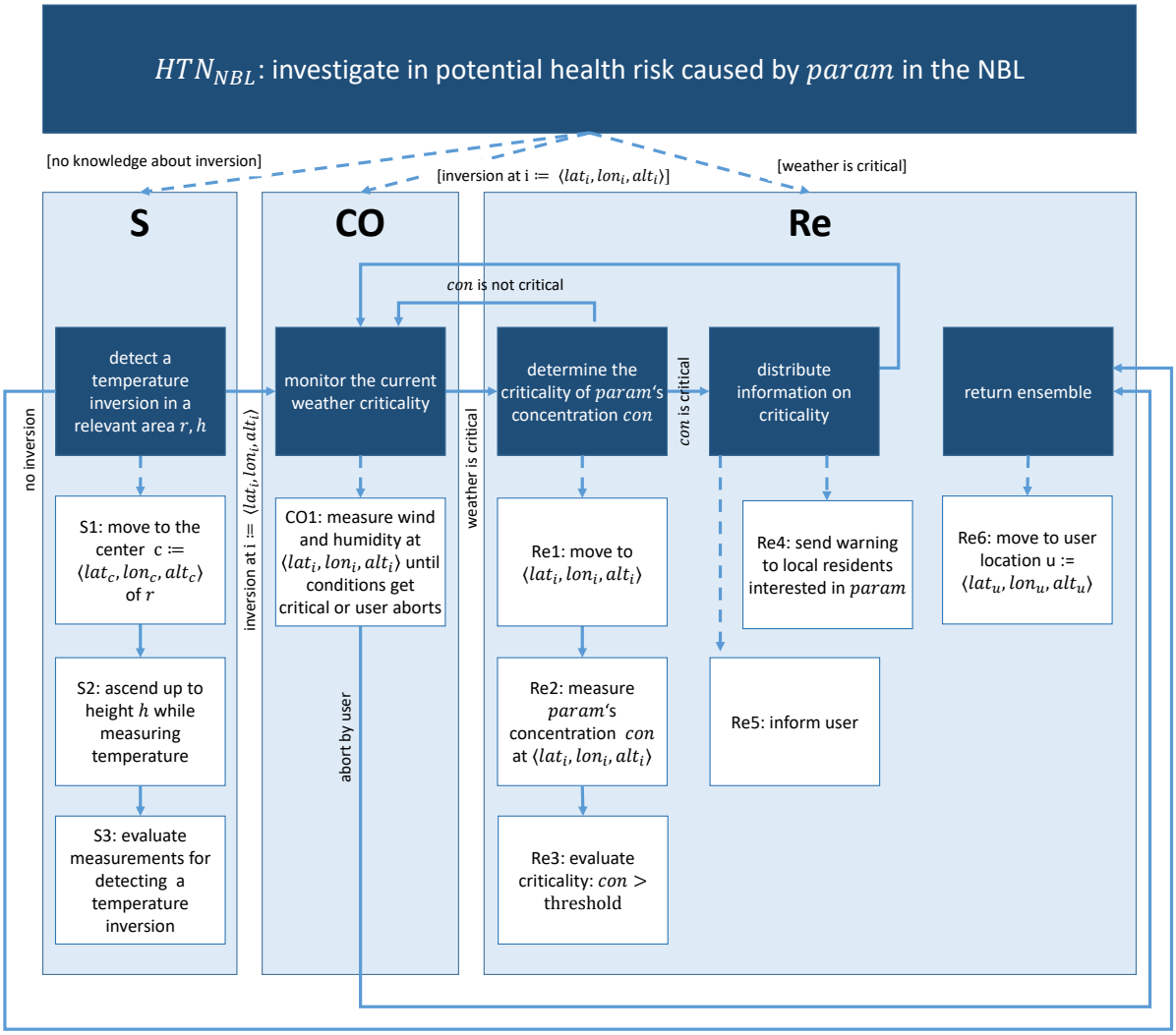


Figure 3.4: Abstract HTN_{NBL} designed for the running example consisting of nodes and directed vertices. Darker colored boxes represent *CNs* containing higher level goals that can be decomposed (dotted vertices) into one or more lower level goals. White boxes represent *PNs* we do not need to further decompose because they already contain concrete commands for acting. Dotted vertices indicate this step of decomposition, i.e., how we can transform *CNs* into partial plans consisting of *PNs* only in this example. In this example, nodes are interconnected with sequential or conditional dependencies (bold vertices) representing the control flow of the HTN_{NBL}. Labels on vertices indicate conditional control flow depending.

instead, we aim at solving those PNs on a collective level. If possible, in its role of a SCORE MISSION DESIGNER (cf. Figure 3.3), the user instructs the ensemble instead of single agents, i.e., generates plans for the ensemble as a whole. This enables the ensemble to compensate for disturbances during the execution of a SCORe mission by itself using the benefits of SO, e.g., by adapting swarm behavior and thus avoids frequent planning.

Running Example: Designing and Initializing the SCORe Mission We now continue our running example introduced in Section 3.2 by describing the necessary content of a SCORe mission definition designed by the user. While we introduce our concrete formalization for mission programming based on HTN in Chapter 4, we give an intuitive and non-technical description here. In its role of a SCORe mission designer (cf. Figure 3.3), we need the user of the Multipotent System to formalize its goal of estimating the impact the current PM concentration can have for residents in a way the Multipotent System $\mathcal{MS}_{\text{NBL}}$ can interpret autonomously.

Following our approach of Combining Planning with Self-Organization, the user thus first needs to design a problem domain description for the current use case. In Figure 3.4, we give an associated, non-technical representation of the problem domain in the form of an informal HTN for our running example's SCORe mission, i.e., HTN_{NBL}. We can find the description of the running example we started in Section 3.2.4 within the darker colored boxes. In the context of HTN [Georgievski and Aiello, 2014], these represent Complex-Nodes *CNs* that need further decomposition into interconnected Primitive-Nodes *PNs* (lighter colored boxes). PNs hold actual commands that can be executed by $\mathcal{MS}_{\text{NBL}}$. During problem domain definition, the user thus needs to propose solutions on how to decompose the topmost goal HTN_{NBL}*: investigate in potential health risk caused by param concentration in the NBL* into goals of lower granularity. Lower-level goals often are interconnected with each other indicating how the ensemble should execute them, e.g., sequential, conditional, or parallel (cf. Figure 3.4). If e.g., we *detect a temperature inversion in a relevant area* r, h, we consequently want to *monitor weather criticality* and so forth. All CNs finally need to be decomposed into interconnected PNs to enable $\mathcal{MS}_{\text{NBL}}$ to generate concrete commands for an ensemble and achieve the topmost goal. E.g., we need to further describe how to reach the goal *detect a temperature inversion in a relevant area* r, h by specifying it as a sequence consisting of the following three steps (cf. Figure 3.4):

1. *move to the center* $c := \langle lat_c, lon_c, alt_c \rangle$ *of* r,
2. *ascend up to height* h *while measuring temperature*, and
3. *evaluate measurements for detecting a temperature inversion.*

While the informal design of the problem domain we propose in Figure 3.4 already includes essential information concerning the necessary control- and data-flow, it does not yet include all required information finally necessary for the execution within $\mathcal{MS}_{\text{NBL}}$. On the one hand, we leave some variables $(param, threshold, r, h)$ unspecified, allowing for the reuse of HTN_{NBL} in other scenarios. For investigating in other critical parameters than PM having other thresholds indicating their criticality *con*, we let variables *param* and *threshold* unassigned in the Complex-Nodes *determine the criticality of param's concentration con* and its decomposition. Likewise, by leaving the area of interest variable concerning the location r and relevant height h in *detect a temperature inversion in a relevant area* r, h, we can use HTN_{NBL} from Figure 3.4 more flexibly. On the other hand, we also do not include concrete specification of requirements concerning the ensemble composition and the concrete implementation of actions in $\mathcal{MS}_{\text{NBL}}$. Whether, e.g., *detect a temperature inversion in a relevant area* r, h finally is executed by a single agent executing a simple measurement flight or a whole ensemble executing an SO

mechanism like, e.g., swarm behavior, we can leave open for $\mathcal{MS}_{\text{NBL}}$ to choose according to its current situation (i.e., concerning how many agents are available in total).

3.2.5.2 Decomposing SCORe Missions into Plans

After defining a SCORe mission in the form of an HTN for the current use-case and for instructing the Multipotent System to work on that mission autonomously, the user can broadcast it to all reachable agents in the Multipotent System (cf. Figure 3.2, where the user can only reach α_1). In their role of MISSION PLANERS (cf. Figure 3.3), agents receiving that SCORe mission start to analyze it and deduce plans for the ensemble according to the current situation of the system (cf. *plan SCORe mission* in Figure 3.2). Because of the relatively low complexity of HTNs in general and their graph-like structure, we can use a slightly adapted depth-first-search [Tarjan, 1972] for searching such a valid plan. If necessary, e.g., if a HTN becomes very detailed or we are interested in optimal plans like Behnke et al. [2019] in their approach, we can support this mechanism with the A* search heuristic approach of Magnenat et al. [2009] keeping the needed time and resources required for planning in a neglectable low range. Thus, the result of that planning is intended to be executed by the Multipotent System, e.g., by instructing single agents to execute individual capabilities or instructing whole ensembles to execute appropriate Collective Capabilities. We intend Collective Capabilities to encapsulate SO-mechanisms like we describe them in Section 3.2.5.1, i.e., to let multiple agents cooperatively work together without the need for explicit instruction and coordination, e.g., like we can achieve it with swarm behavior. To exploit the beneficial properties of robustness and scalability, we aim to decompose SCORe missions formulated as HTN into plans the Multipotent System can execute on the collective level with Collective Capabilities whenever possible. Nevertheless, we cannot wholly avoid explicitly planned alternatives for solving the goals formulated in an HTN. This is necessary at least as a fallback solution if there are not enough agents available to realize the individual collective behavior, e.g., a specific swarm behavior we encapsulate in a Collective Capability. In other situations, it can also better suffice the Multipotent System's user's needs to instruct individual agents explicitly. In Figure 3.4, for executing *detect a temperature inversion in a relevant area* r, h, we can either use a group of agents executing a coordinated movement using an appropriate swarm algorithm covering a larger area and potentially detecting an existent temperature inversion in a better way (we further investigate in executing such Collective Capabilities in Chapter 7), or rely on a single agent if there are not enough agents available in $\mathcal{A}_{\mathcal{MS}}$.

Running Example: SCORe Mission Planning We continue our running example further illustrating the process of generating a concrete plan for the HTN_{NBL} defined in Figure 3.4. Taking a closer look at the Complex-Nodes CNs including their decomposition, we can identify the three phases of the SCORe mission, each requiring differently composed ensembles for working on them.

1. We identify a partial plan PART-$\rho_{\text{NBL-S}} \subset HTN_{\text{NBL}}$ designed to Search the temperature inversion, requiring a flying ensemble $\mathcal{E}^{\text{NBL-S}}$ consisting of agents only that can also measure temperature, i.e.,

$\forall\, \alpha \in \mathcal{E}^{\text{NBL-S}} \subseteq \mathcal{A}_{\text{NBL}}$: $\{\ c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-TEMP}}\ \} \subseteq \mathcal{C}_\alpha$.

2. We identify a partial plan PART-$\rho_{\text{NBL-CO}} \subset HTN_{\text{NBL}}$ designed to Continuously Observe weather conditions, requiring a flying ensemble $\mathcal{E}^{\text{NBL-CO}}$ consisting of agents only that can also measure weather, i.e.,
   $\forall\, \alpha \in \mathcal{E}^{\text{NBL-CO}} \subseteq \mathcal{A}_{\text{NBL}}$: $\{\ c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-WIND}}, c^p_{\text{M-HUM}}\ \} \subseteq \mathcal{C}_\alpha$.
3. We identify a partial plan PART-$\rho_{\text{NBL-RE}} \subset HTN_{\text{NBL}}$ designed to React to critical weather conditions, requiring a flying ensemble $\mathcal{E}^{\text{NBL-RE}}$ consisting of agents only that can also measure *param*, i.e.,
   $\forall\, \alpha \in \mathcal{E}^{\text{NBL-RE}} \subseteq \mathcal{A}_{\text{NBL}}$: $\{\ c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-PM}}\ \} \subseteq \mathcal{C}_\alpha$.

Thus, an ensemble $\mathcal{E}^{\text{NBL-SCORE}}$ capable of solving all three phases of the SCORe mission as a whole would require agents providing all capabilities required in any of the three phases of the SCORe mission, i.e.,

$\forall\, \alpha \in \mathcal{E}^{\text{NBL-SCORE}} \subseteq \mathcal{A}_{\text{NBL}}$: $\{\ c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-WIND}}, c^p_{\text{M-HUM}}, c^p_{\text{M-PM}}\ \} \subseteq \mathcal{C}_\alpha$.

Our exemplary ensemble $\mathcal{MS}_{\text{NBL}} = (\mathcal{A}_{\text{NBL}}, \mathcal{SDH}_{\text{NBL}})$ in mind, due to the weight restrictions introduced by real hardware $\mathcal{SDH}_{\text{NBL}}$, we can see that there is no possible configuration of the $\mathcal{MS}_{\text{NBL}}$ offering such agents. While our approach aims at solving such problematic situations by bringing into play the property of being multipotent, i.e., being able to adapt its configuration when needed, the user needs to provide information to the system *when* to play this potential avoiding too frequent and possibly time-consuming physical reconfiguration. We achieve this by offering the user possibilities to include triggers in an HTN explicitly enabling the Multipotent System for reconfiguring its composition if required in the form of Replanning-Nodes. Typically, we can apply planning when switching from one phase of a SCORe mission to another. This results in plans describing necessary actions of only one phase of a SCORe mission, making it easier to identify and form an ensemble for solving it. Thus, we make use of this potential and focus on the the first phase of Search in HTN_{NBL}, i.e., PART-$\rho_{\text{NBL-S}}$, for the moment and come back again to the other phases of the SCORe mission later in the course of our running example: Planning on the HTN_{NBL} returns an initial plan $\rho_{\text{NBL-S}}$ containing the partial plan PART-$\rho_{\text{NBL-S}}$ from Figure 3.4 combined with a subsequent action:

$$\rho_{\text{NBL-S}} := [\text{S1}, \text{S2}, \text{S3}, \begin{cases} \text{RE6}] & \text{if NO INVERSION,} \\ \text{PLAN: } HTN_{\text{NBL}}] & \text{if INVERSION AT } i, \end{cases}$$

We find the plan $\rho_{\text{NBL-S}}$ when we decompose our topmost goal in HTN_{NBL}*: investigate in potential health risk caused by param in the NBL* according to the current knowledge about the world's state (we assume we have no knowledge about an inversion at the beginning of the SCORe mission) and stop planning when detecting a change in the phase of the SCORe mission (cf. Figure 3.5).

3.2.5.3 Selecting, Activating, and Finalizing Plans

After one agent has achieved a valid decomposition from a HTN by planning, i.e., a plan candidate, this plan candidate is distributed within the Multipotent System following the dependencies retrieved from the plan (cf. *select plan* in Figure 3.2). Because this step can be done asynchronously and in parallel by all agents receiving the HTN, the Multipotent System

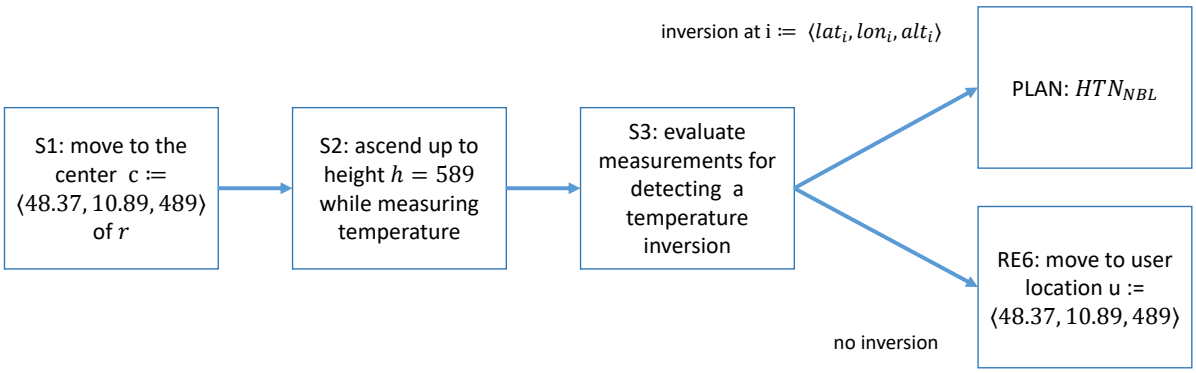


Figure 3.5: Informal plan $\rho_{\text{NBL-S}}$ resulting from initial planning with HTN_{NBL}. Parameters for the area of interest r, the maximum height for evaluation h, and the location of the user u were set by the user before planning: $r := \langle 48.37, 10.89, 489\rangle$ (i.e., the geographic coordinates including the ASL of the center of Augsburg City), $h := 589$ (i.e., the height calculated as the ASL added up with 100 meters), and $u := \langle 48.37, 10.89, 489\rangle$ (i.e., the user's location is the same as the center of r).

possibly has to synchronize planning results and collectively determine one solution to be executed. This happens in *activate plan* in Figure 3.2), e.g., by letting the agents perform leader election in their roles of LEADER VOTERS (cf. Figure 3.3) within the set of agents delivering plan candidates. For leader election, we can use well-established mechanisms like that proposed in the survey of Gharehchopogh and Arjang [2014], e.g., the often adopted Raft algorithm [Huang et al., 2020], which we do not require to adapt specifically for Multipotent Systems. After deciding on a plan cooperatively, SCORe missions and their phases occurring in plans are designed to be enriched with parameters, enabling their applicability in different situations and completely different use-cases in principle. For example, cooperatively searching for a specific parameter in a defined area can be parametrized regarding the concerned area, the affected parameter, or a quality requirement (e.g., needed measurement accuracy). Those parameters and requirements must be defined in the HTN describing the respective SCORe mission beforehand. Thus, the user needs to design them accordingly so they can be included in plans that are finally executed by the Multipotent System (cf. r, h, p in our running example). The coordinated execution of SCORe missions then takes place on the ENSEMBLE LAYER and its subordinate AGENT LAYER and SEMANTIC HARDWARE LAYER. On PLAN LAYER, the agent winning the leader election therefore adopts the role of a PLAN LEADER for the respective plan (cf. Figure 3.3). After a plan is wholly worked off, the respectively formed ensemble informs the system's user about the state of execution concerning the introduced SCORe mission (cf. *set plan status finalized* Figure 3.2). According to the descriptions from the plan derived from the HTN, the ensemble is then allowed to autonomously continue executing the SCORe mission, i.e., by performing subsequent planning using the results of the current plan's execution or return to an idle state waiting for further instructions.

Running Example: Select, Activate, and Finalizing Plans To illustrate this interplay, we continue our running example. Again, we regard our Multipotent System $\mathcal{MS}_{\text{NBL}}$ now initially facing the challenge to plan with HTN_{NBL} (cf. Figure 3.5). For the example of the SCORe mission HTN_{NBL}, we assume the user reaches α_2 when introducing HTN_{NBL}. Agent α_2 then adopts the role of a MISSION PLANER OF HTN_{NBL} (cf. Figure 3.6). The agent α_2 thus distributes HTN_{NBL} to α_1 and α_3. The new knowledge available to every agent then triggers them to adopt the additional role of a LEADER VOTER FOR HTN_{NBL}. In these roles, α_1, α_2, and α_3 then start a leader election process for determining the leader of HTN_{NBL}, i.e., which agent should create a plan for HTN_{NBL}. In the example, α_2 wins the election, creates a plan $\rho_{\text{NBL-S}}$ for the SCORe mission HTN_{NBL}, thus adopts the role of the PLAN LEADER FOR $\rho_{\text{NBL-S}}$ (cf. Figure 3.5), and continues the process of working on that plan on ENSEMBLE LAYER, which we focus on later. When the plan is finally worked through by the ensemble, the PLAN LEADER FOR $\rho_{\text{NBL-S}}$, i.e., agent α_2, gets informed by the ENSEMBLE LAYER. Then, α_2 can finalize the plan $\rho_{\text{NBL-S}}$, i.e., set its status and results in the world state WS accordingly and evaluate whether the SCORe mission the plan originated from requires further steps to be executed. If during $\rho_{\text{NBL-S}}$ no temperature inversion was detected by the ensemble working on $\rho_{\text{NBL-S}}$, the agent can finalize the plan because $\rho_{\text{NBL-S}}$ does not include further steps to be executed by the PLAN LEADER FOR $\rho_{\text{NBL-S}}$. If otherwise, a temperature inversion was detected at location i, finalizing $\rho_{\text{NBL-S}}$ requires the PLAN LEADER FOR $\rho_{\text{NBL-S}}$ to perform subsequent planning using HTN_{NBL} and an appropriately updated world state WS. Planning thus generates a new plan, i.e., the plan $\rho_{\text{NBL-CO}}$ dedicated at *Continuously Observing* the temperature inversion, requiring another ensemble for its execution. By integrating planning steps in plans $\rho_{\text{NBL-S}}$, $\rho_{\text{NBL-CO}}$, and $\rho_{\text{NBL-RE}}$, we can achieve the Multipotent System $\mathcal{MS}_{\text{NBL}}$ to entirely execute the whole SCORe mission defined in HTN_{NBL} successively and, if necessary, adapt its configuration respectively.

3.2.6 The Ensemble Layer

As we described in Section 3.2.5, we want to be able to formulate instructions in Primitive-Nodes of HTN on the collective level whenever possible and intend ensembles to execute respective plans consisting of such Primitive-Nodes. This design decision raises the need for appropriate actions we must define on the ensemble level. Actions have to assure that the execution of associated nodes produces the post-condition the plan relies on. Thus, for every phase of a SCORe mission, we need to provide appropriate actions for solving it on the collective level. Therefore, we introduce *Ensemble Programs*. Ensemble Programs encapsulate the general logic for the cooperative and coordinated execution of the associated SCORe phase. Where possible, we enrich those Ensemble Programs with Collective Capabilities encapsulating SO-mechanisms, e.g., adapting swarm behavior where the interplay of local actions executed on the subordinate AGENT LAYER (cf. Section 3.2.7) produces the desired effect as the result of emergence as proposed by Goldstein [1999] on the ENSEMBLE LAYER. For executing Ensemble Programs, the ENSEMBLE LAYER and AGENT LAYER have to work in a highly integrated way. We call the procedures needed for coordinating Ensemble Programs on ENSEMBLE LAYER *ensemble level parts* and the procedures needed for cooperatively executing the Ensemble Programs on AGENT LAYER *agent level parts*. We generate these program parts from the plan resulting from automated planning performed on PLAN LAYER, i.e., derive coordination and cooperation instructions directly from the plan (cf. *generate ensemble program* in Figure 3.2). For achieving

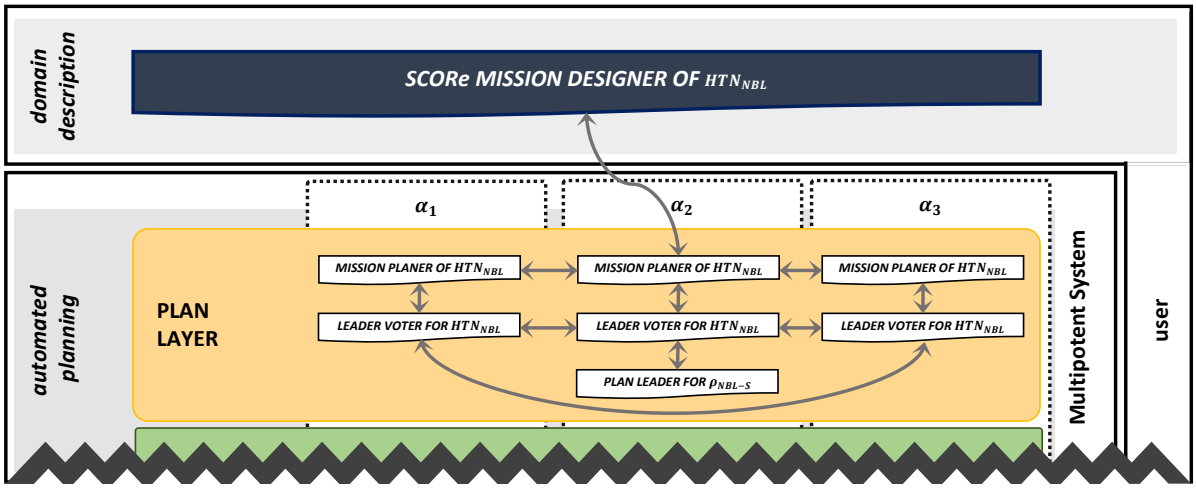


Figure 3.6: Roles adopted by agents α_1, α_2, and α_3 on the PLAN LAYER from our running example. We focus on the interplay of the user in the role of a SCORE MISSION DESIGNER and the PLAN LAYER of the Multipotent System during the initialization process performed by the user (because other layers do not influence the process, we thus skip them here). The user introduces a new SCORe mission designed for its current use case by communicating it to one agent of the Multipotent System adopting the role of a MISSION PLANER (cf. Figure 3.3). The agent receiving the new SCORe mission defined in a HTN then distributes its knowledge to all other agents in the Multipotent System it can reach, i.e., addresses other MISSION PLANNERS. In their roles of LEADER VOTERS, agents then elect a leader for HTN_{NBL}. The winner of this election then adopts the role of the PLAN LEADER for the plan, e.g., PLAN LEADER FOR $\rho_{\text{NBL-S}}$.

the actual execution of Ensemble Programs, we then rely on the AGENT LAYER implementing the agent level parts to exploiting local interactions in an SO manner, coordinated by the ensemble level part if necessary. For processing the results of such Ensemble Programs, we again use the ensemble level parts. Besides the execution of plans, also structural coordination in the form of Ensemble Formation and configuration (cf. *form ensemble*, *allocate ensemble program* in Figure 3.2), needs to be initiated and controlled on the ENSEMBLE LAYER.

3.2.6.1 Mapping Plan Requirements to Ensemble Programs

One key element for providing SO-execution of Ensemble Programs are Collective Capabilities encapsulating SO-mechanisms like swarm behavior. To be applicable not only for one specific instance of Search, Continuously Observe, or React but for the whole class of that phase instead, we intend Collective Capabilities to be parameterizable. This way, Collective Capabilities fit the concept of the also parameterizable Primitive-Nodes that HTN can consist of (cf. Section 3.2.5). Many Search phases of different SCORe missions, e.g., can be handled by the same Collective Capability, only varying in the parameter of interest. We can handle searching for, e.g.,

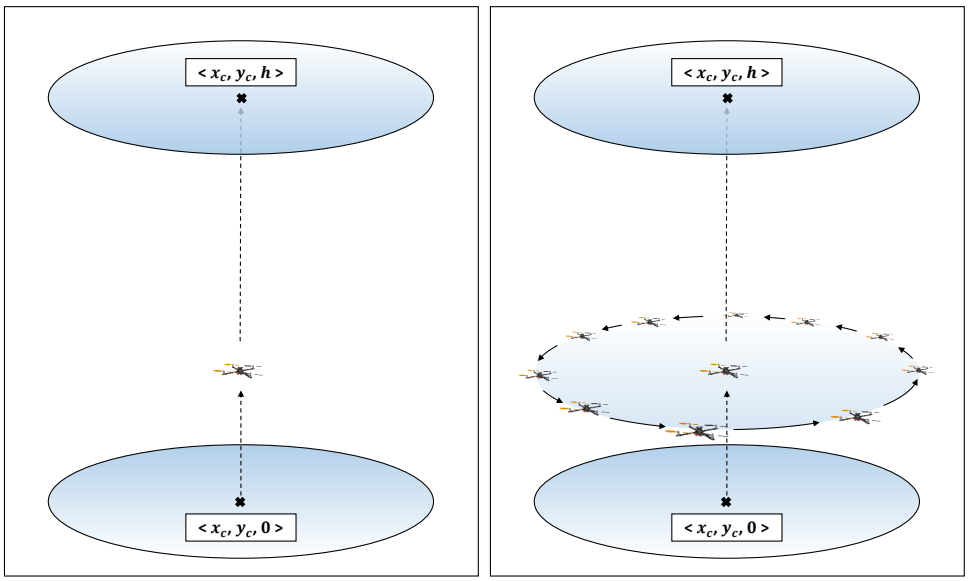
- the source of a specific gas X in gas accidents like we describe in the case study on *Dealing with Chemical Accidents* Section 2.5,

- the highest temperature in case of fire accidents (cf. case study on *Dealing with Forest Fires* in Section 2.6),
- the highest PM concentration like we are interested in in the case study on *Improving Climate Models* in Section 2.3,
- or any other parameter with a continuous distribution characteristic

by letting the ensemble execute the same Collective Capability when we adapt the Particle Swarm Optimization (PSO) algorithm like proposed by Zhang et al. [2015]. If the ensemble level part and the agent level parts from the Ensemble Program only address such Collective Capabilities, it only requires the adaptation and parametrization of the respective local rules. In the case of PSO, these local rules in agent level parts consist of a program equences requiring the repeated execution of measuring the parameter of interest, communicating it to the other agents, and adapting the current trajectory of the agent. Because PSO is a concrete instance of a swarm algorithm relying on the cooperation of multiple agents for producing the emergent effect, the execution of the PSO is suitable in situations where there are enough of those agents available. Suppose this criterion is fulfilled within the ensemble, and every agent in the ensemble is working according to the local rules of the PSO (cf. the description in Section 3.2.7). In that case, the ensemble achieves the objective of finding a global maximal concentration of the parameter of interest as the emergent effect. Other phases of SCORe missions may require adopting other swarm behavior in a Collective Capability, e.g., movements of the whole ensemble with guided boiding (i.e., control one, move all) [Vásárhelyi et al., 2014] in React phases of SCORe missions, or adapted potential field mechanisms [Koren and Borenstein, 1991] in Continuously Observe phases of SCORe missions, among others. We elaborate on this and other local rules used in Collective Capabilities in Chapter 7.

However, applying SO-mechanisms like swarm algorithms is not appropriate in every situation (e.g., when we require to move a single agent to a predefined location), not even possible to determine (e.g., in specific React phases of a SCORe mission requiring actions from a single agent only), or does not fit to the current conditions the system is situated in (e.g., the desired effect can not be produced because too few agents are available). As we describe in Section 3.2.5, we therefore also offer the possibility to fall back to classical MAS/MRS planning for creating plans when required and thus also support generating Ensemble Programs including explicit instructions on ENSEMBLE LAYER. The same way we avoid designing plans containing instructions for individual agents, we also aim to avoid Ensemble Programs to contain such individual instructions whenever possible and use Collective Capabilities instead, enabling us to exploit the emergent effects they can encapsulate. With this strategy, we do not lose possibilities in controlling the ensemble explicitly if necessary but can exploit the advantage SO-mechanisms have over pure MRS- / MAS-Systems when appropriate.

Additionally, we exploit situations where multiple solutions for solving the same phase of a SCORe mission are available according to the current situation within the ensemble. For example, accomplishing the Search for the highest concentration of a parameter of interest can be achieved in different ways by the Multipotent System: *a)* by executing PSO encapsulated in a Collective Capability if there are enough agents available in the system for producing the required emergent effect, or *b)* by executing a systematic search with a reduced amount of agents requiring planning on the level of individual agents. The alternatives require different degrees of coordination among agents in the ensemble to be defined in respective Ensemble Programs. If a user includes such alternatives during the design of SCORe missions in a HTN, we can let the Multipotent System exploit this additional degree of freedom in decisions



(a) using a single agent (b) using a swarm algorithm

Figure 3.7: Determining the occurrence of a temperature inversion using an ensemble $\mathcal{E}^{\text{NBL-S}}$ consisting of only one agent (Figure 3.7a) as we describe it in detail in our running example or a group of agents (Figure 3.7b) executing the Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior (cf. Chapter 7 for more details on its execution) in a plan $\rho_{\text{NBL-S}}$.

during run-time for increasing the efficiency and the robustness of execution if possible. The agent adopting the role of a ENSEMBLE COORDINATOR (cf. Figure 3.3) can make the decision to apply a particular solution by using a respective plan derived from the SCORe mission encoding the possible alternatives (cf. *select SCORe plan* in Figure 3.2) and dependent on, e.g., the current amount of agents available for executing it or other run-time parameters. Such run-time parameters can be set during a previous execution of the same or other plans and thus can impact the decisions in the current plan. If the user-specified such alternatives in a plan, the executing ensemble thus can autonomously choose the best option at run-time. That way, we can generate Ensemble Programs offering precisely tailored alternatives to the respective conditions the ensemble is situated in. That way, we can generate Ensemble Programs offering precisely tailored alternatives to the respective conditions the ensemble is situated in. We investigate that topic in more detail in Chapter 4 describing how to design Ensemble Programs and generate the respective ensemble level part and the agent level parts containing such alternatives. In Chapter 7, we then describe how to execute Ensemble Programs.

Running Example: Generating the Ensemble and agent level parts We continue our running example describing how we can generate Ensemble Programs consisting of an ensemble level part and agent level parts from the plan $\rho_{\text{NBL-S}}$ derived from the SCORe mission defined in HTN_{NBL}. Looking at Figure 3.5, we can identify three Primitive-Nodes in $\rho_{\text{NBL-S}}$ encapsulating information necessary for agents working on the plan in the respective ensemble $\mathcal{E}^{\text{NBL-S}}$: S1, S2, and RE6 all hold information commanding the execution of capabilities, i.e., S1

requires executing $c^p_{\text{MV-POS}}$, S2 requires executing $c^p_{\text{MV-VEL}}$ and $c^p_{\text{M-TEMP}}$, RE6 requires executing $c^p_{\text{MV-POS}}$. The other two nodes, S3 and PLAN, encapsulate coordination information necessary for agents coordinating the ensemble $\mathcal{E}^{\text{NBL-S}}$ instead: S3 encapsulates instructions for aggregating the information derived during executing S2 and PLAN encapsulates information on how to further progress within the SCORe mission after evaluating the result of S3 (i.e, whether the ensemble detected a temperature inversion or not).

Thus, when we want to provide the possibility for the executing ensemble $\mathcal{E}^{\text{NBL-S}}$ to autonomously decide whether to apply a Collective Capabilities, i.e., alternative (b) in Figure 3.7b, or fall back to a plan commanding individual agents, i.e., alternative (a) in Figure 3.7a, we need to provide alternative execution patterns for S1, S2, and RE6 and a respective coordination pattern for S3. Because PLAN in $\rho_{\text{NBL-S}}$ only depends on whether a temperature inversion was detected in a previous step, we do not need to provide alternatives here.

Because we generate the central information in $\rho_{\text{NBL-S}}$ within the node S2, we can ideally find a Collective Capability implementing it. For the running example, we can find such by applying the idea of a flying ensemble replicating the remote sensing technology of 3D Doppler boundary layer lidar towers originating from Case Study 2 (cf. Figure 2.5b): For determining the spatial distribution of the temperature inversion with higher quality and reliability than that a single agent can provide, we might want to use a whole group of agents instead (cf. Section 2.4). By using a swarm algorithm producing a *Ring-of-Fliers* (cf. Chapter 7 for more details on its implementation) implemented by a Collective Capability, we can command a swarm of agents to collectively move in a coordinated and synchronized way while measuring a parameter of interest. We can parameterize the Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior to let the ensemble $\mathcal{E}^{\text{NBL-S}}$ measure temperature while simultaneously ascending to the required height h, permanently synchronizing the altitude of all participating agents. By aggregating and evaluating the information generated by the ensemble in S3, we can derive much more reliable data concerning the occurrence of a temperature inversion, which might be of interest for the user of $\mathcal{MS}_{\text{NBL}}$ due to its effect within the course of the whole SCORe mission defined in HTN_{NBL}. To let $\mathcal{MS}_{\text{NBL}}$ decide which option to choose during runtime requires an indicator. We can use the number of available agents here. This number can either be defined directly by the respective swarm algorithm, i.e., the Collective Capability is aware of a minimum amount of agents necessary for producing the emergent effect, or the designer of HTN_{NBL} defines that number. In our running example, we assume a user-defined minimum amount of 5 agents for executing the Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior. Table 3.3 provides the necessary role adoptions in $\mathcal{E}^{\text{NBL-S}}$ for both alternatives (a) and (b). We see, alternatives (a) and (b) both have the same requirements concerning capabilities an agent agents must provide for participating in $\mathcal{E}^{\text{NBL-S}}$, i.e.,

$\forall a \in \mathcal{E}^{\text{NBL-S(A)}}, \mathcal{E}^{\text{NBL-S(B)}} : \{\ c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-TEMP}}\ \} \in C_a$.

When regarding our Multipotent System $\mathcal{MS}_{\text{NBL}}$ facing the plan $\rho_{\text{NBL-S}}$ (cf. Figure 3.5) derived by autonomous planning from the HTN_{NBL} (cf. Figure 3.4), we can see that there is no sufficient number of agents available for successfully applying alternative (b) in that situation. Thus, for the further course of the running example in this Chapter, we rely on the alternative (a) involving particular agents instead of Collective Capabilities. This keeps the example clearer and easier to understand regarding the technologies on ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER. Nevertheless, we provide information on applying the alternative using a Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior where relevant.

Table 3.3: Necessary actions performed by agents adopting different roles in $\mathcal{E}^{\text{NBL-S}}$ for alternatives (a) and (b) in the running example.

node	alternative (a): $\mathcal{E}^{\text{NBL-S(A)}}$	alternative (b): $\mathcal{E}^{\text{NBL-S(B)}}$
S1	one PLAN WORKER (WORK_1) executing $c^p_{\text{MV-POS}}$ to $\langle lat_c, lon_c, alt_c \rangle$	all PLAN WORKERs of $\mathcal{E}^{\text{NBL-S(B)}}$ (WORK_s), executing $c^p_{\text{MV-POS}}$ to $\langle lat_c, lon_c, alt_c \rangle$
S2	WORK_1 executing $c^p_{\text{MV-VEL}}$ up to height h	swarm ≥ 5 WORK_s executing *Ring-of-Fliers* to height h
S3	PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$ (COORD) evaluating result of WORK_1	PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$ (COORD) evaluating result of all WORK_s
PLAN	COORD using the result of S3	COORD using result of S3
RE6	WORK_1 executing $c^p_{\text{MV-POS}}$ to $\langle lat_u, lon_u, alt_u \rangle$	WORK_s executing $c^p_{\text{MV-POS}}$ to $\langle lat_u, lon_u, alt_u \rangle$

3.2.6.2 Forming Ensembles by Allocating agent level parts

One fundamental responsibility of the ENSEMBLE LAYER to correctly execute a plan ρ is to determine an appropriate ensemble $\mathcal{E}^\rho$ for working on ρ while respecting all requirements of ρ regarding its coordination and execution. This process is influenced by the respective conditions in the Multipotent System, e.g., the number of agents in the system and their respective configuration. Shehory and Kraus [1998] propose this problem can be solved by task allocation achieved through coalition formation. In the context of Multipotent System, we can use this finding by looking at the problem of Ensemble Formation as an instance of such of coalition formation:

For a specific plan, the ensemble level part of a Ensemble Program then contains the relevant information for coordinating the respective plan (coordination information $\mathcal{CI}^\rho$). The agent level parts in a Ensemble Program instead contain information on the requirements for participating in that plan. Because different agent level parts in a plan ρ may differ from each other concerning their requirements and execution, we encapsulate both information combined in a task t each. That way, we can define which capabilities $\mathcal{C}_t$ are needed to execute the agent level part and how to cooperate with other agents from the ensemble during the execution in $\mathcal{CP}^\rho_t$. Thus, for each plan ρ, there exists a set of tasks $t \in \mathcal{T}^\rho$, we require to be allocated to agents each for working on the ρ. The set of agents fulfilling these requirements then form the ensemble $\mathcal{E}^\rho$ for that plan.

Agents in an ensemble created by Ensemble Formation can work on the respective plan cooperatively with the algorithm best fitting to the current situation. In our layered reference architecture, we can form ensembles by appropriate interactions between ENSEMBLE LAYER (implementing the activities *form ensemble* and *allocate ensemble program* in Figure 3.2) and AGENT LAYER (implementing the activity *participate if executable* on AGENT LAYER in Figure 3.2). For solving the problem of Ensemble Formation, we propose the use of a market-based approach often applied to MAS/MRS [Zlot and Stentz, 2006; Dias et al., 2006; Hussein et al., 2014; Kosak et al., 2015] for distributing computation requirements in the Multipotent System problem-specific, for reducing communication overhead when not required, and for exploiting

local information available at individual agents. For forming an ensemble dedicated to handling a specific plan, one agent needs to act in the role of a PLAN AUCTIONEER adopted on its ENSEMBLE LAYER and communicate with other agents in the ensemble that adopt the role of a PLAN BIDDER on their respective AGENT LAYER each (cf. Figure 3.3). The agent adopting the role of a PLAN AUCTIONEER of a specific plan on ENSEMBLE LAYER then is the same agent that also adopts the role of the PLAN LEADER on PLAN LAYER. By initiating one Call for Proposals (CfP) for the respective plan, a PLAN AUCTIONEER tries to find enough PLAN BIDDERs to ensure the plan's requirements. PLAN BIDDERs answer that Call for Proposals (CfP) after evaluating their qualification for participating in the plan. Therefore, they evaluate whether they meet the defined requirements defined in the respective tasks $t \in \mathcal{T}^{\rho}$ or not (cf. Section 3.2.7). This split-up of responsibilities avoids a single point of failure a centralized version would suffer from. According to Anders et al. [2016], decentralization prevents problems caused by the spontaneous breakdown of agents (when an agent does not respond to the PLAN AUCTIONEER, its qualification can be seen as nonexistent) and parallelizes computationally expensive activities. It further better fits the way information is distributed in the Multipotent System: When agents propose their qualification for working on a plan, we use the relevant knowledge exactly where we process it, i.e., locally within the agent using its self-awareness capabilities. Otherwise, we would require a complex information synchronization mechanism when solving the problem of Ensemble Formation centrally. Like illustrated in Figure 3.2, the activity of *allocate Ensemble Program* can only be achieved if all requirements defined in the plan are fulfilled. Using the market-based approach for Ensemble Formation, we can take care of this in a distributed fashion. We investigate in the details of our approach for Ensemble Formation accompanied with complex examples in a dedicated chapter (cf. Chapter 5).

Running Example: Ensemble Formation for the Plan $\rho_{\text{NBL-S}}$ We continue our running example to illustrate the process of allocating the tasks $\mathcal{T}^{\rho}$ contained in $\rho_{\text{NBL-S}}$ within the Multipotent System $\mathcal{MS}_{\text{NBL}}$. Because α_2 became the leader of plan $\rho_{\text{NBL-S}}$, it also adopts the role of the plan's auctioneer (cf. PLAN AUCTIONEER OF $\rho_{\text{NBL-S}}$ in Figure 3.8). In the fashion of a market-based approach, it sends one CfP to all AGENT LAYERs of agents $\alpha \in \mathcal{A}_{\text{NBL}}$ it can reach, i.e., to α_1 and α_3 but also α_2 (agent α_2 thus can also participate in the ensemble it already coordinates). Consequently, all agents receiving this CfP adopt the role of a PLAN BIDDER IN $\rho_{\text{NBL-S}}$, bidding on the tasks $t \in \mathcal{T}^{\rho}$. To determine whether they can participate in the ensemble $\mathcal{E}^{\text{NBL-S}}$, agents α_1, α_2, and α_3 evaluate their qualification regarding the requirements concerning capabilities encoded in the tasks of $\rho_{\text{NBL-S}}$ and return a proposal for participation to the respective request. According to the number of responses α_2 receives in the role of the PLAN AUCTIONEER OF $\rho_{\text{NBL-S}}$, α_2 then decides on an alternative of $\rho_{\text{NBL-S}}$, i.e., alternatives (a) or (b) in our example. Subsequently, agent α_2 generates the respective ensemble level part (containing the relevant coordination information $\mathcal{CI}^{\rho}$) and agent level parts (containing the relevant cooperation pattern $\mathcal{CP}^{\rho}_{t}$) belonging to the Ensemble Program generated from $\rho_{\text{NBL-S}}$ and allocates them to the respective agents. For the Multipotent System $\mathcal{MS}_{\text{NBL}}$ we look at in this running example, agents α_2 and α_3 can provide all required capabilities, while α_1 cannot ($c^{p}_{\text{M-TEMP}} \notin C_{\alpha_1}$).

Because in our simplified running example α_2 does not receive enough proposals for creating an ensemble using alternative (b) exploiting the emergent effect of a Collective Capability (only

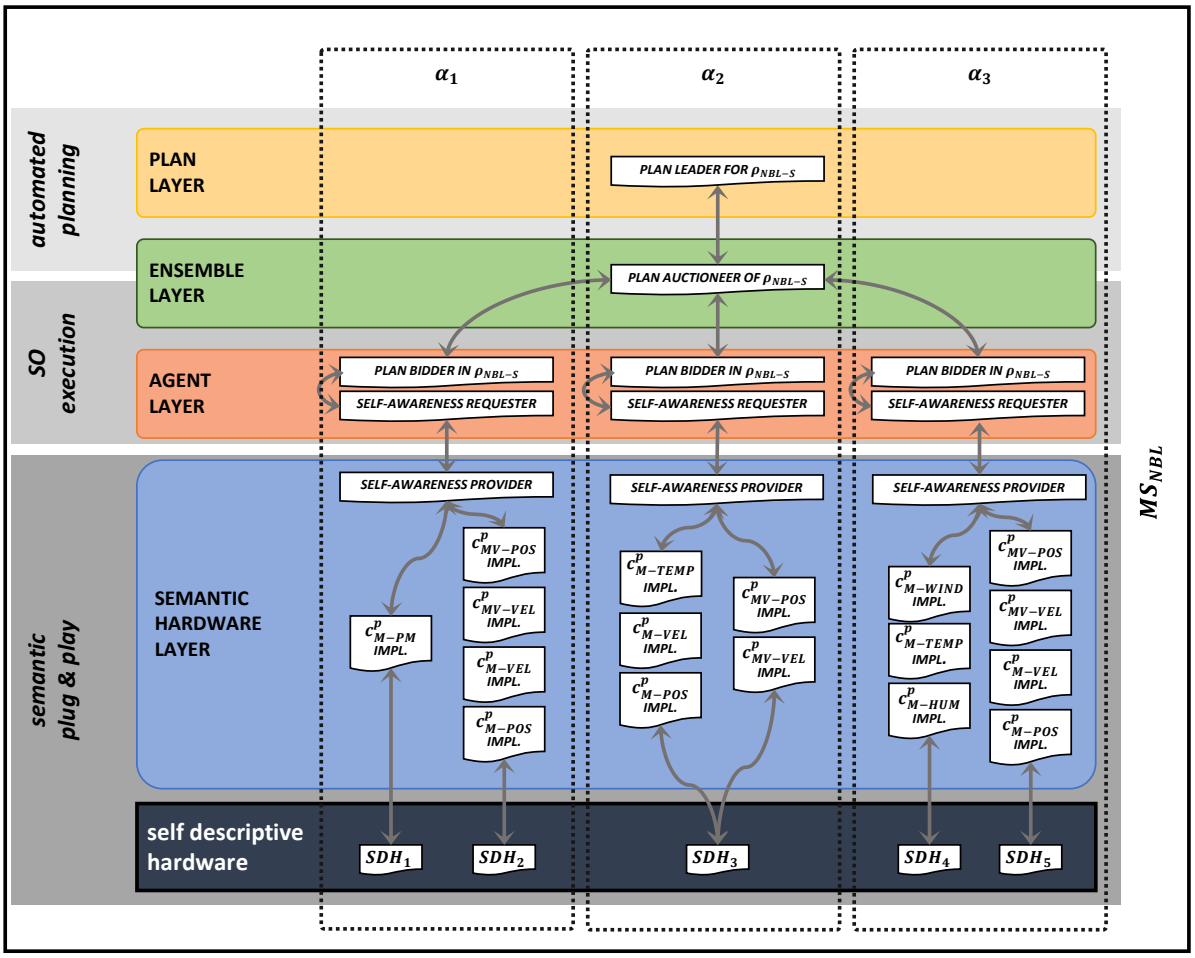


Figure 3.8: Concrete instantiation of the Multipotent System $\mathcal{MS}_{\text{NBL}}$ with agents $\{\alpha_1, \alpha_2, \alpha_3\} = \mathcal{A}_{\text{NBL}}$ and $\{\text{SDH}_1, \text{SDH}_2, \text{SDH}_3, \text{SDH}_4, \text{SDH}_5\} = \mathcal{SDH}_{\text{NBL}}$ during the *allocation* of plan $\rho_{\text{NBL-S}}$ after α_2 became the PLAN LEADER FOR $\rho_{\text{NBL-S}}$ on PLAN LAYER and that of the PLAN AUCTIONEER OF $\rho_{\text{NBL-S}}$ on ENSEMBLE LAYER. All agents $\alpha \in \mathcal{A}_{\text{NBL}}$ participate in the process of Ensemble Formation by adopting the roles of PLAN BIDDER IN $\rho_{\text{NBL-S}}$. For determining their qualification of participating in the ensemble $\mathcal{E}^{\text{NBL-S}}$, in their role of PLAN BIDDER IN $\rho_{\text{NBL-S}}$ agents request relevant information concerning capabilities they provide from the SEMANTIC HARDWARE LAYER. Therefore, agents adopt the role of SELF-AWARENESS REQUESTER on AGENT LAYER exchanging information with SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER. The SELF-AWARENESS PROVIDER then can analyze the current local situation concerning the available and executable capabilities relevant for bidding on all tasks $t \in \mathcal{T}^{\rho}$ for $\rho_{\text{NBL-S}}$. Because, e.g., agent α_2 is configured with SDH_3 it can provide and execute capabilities $c^p_{\text{M-TEMP}}$, $c^p_{\text{M-POS}}$, $c^p_{\text{M-VEL}}$, $c^p_{\text{MV-POS}}$, $c^p_{\text{MV-VEL}}$ and thus adopts the respective roles of CAPABILITY IMPLEMENTERS on SEMANTIC HARDWARE LAYER.

two instead of five required proposals), it needs to fall back to form the ensemble for alternative (a) using the explicitly formulated version consisting of individual instruction for agents. Thus, α_2 needs to decide between proposals received from α_1 and α_2 for allocating the agent level part because the alternative (a) only requires one agent adopting the role of a PLAN WORKER IN $\rho_{\text{NBL-S}}$. For the sake of simplicity in our running example, we assume agent α_2 selects the proposal itself made in the role of a PLAN BIDDER OF $\rho_{\text{NBL-S}}$ and thus also adopts the role of the required PLAN WORKER IN $\rho_{\text{NBL-S}}$ on AGENT LAYER itself (cf. Figure 3.9). To compensate for the simplicity of our running example, we provide complex examples using Collective Capabilities and larger ensembles in the detailled explanations in Chapter 5, Chapter 7.

3.2.6.3 Executing Ensemble Programs by Coordinating Agent Level Parts with the Ensemble Level Part

If the agent adopting the role of the PLAN AUCTIONEER (cf. Figure 3.3) can find an ensemble $\mathcal{E}^\rho$ fulfilling all requirements of ρ, it can activate the plan on the PLAN LAYER (cf. the activity *activate plan* in Figure 3.2) in its role of the PLAN LEADER. Subsequently, in its role of an PLAN COORDINATOR, the agent then coordinates the execution of the associated agent level parts (cf. the activities *run ensemble level part* and *run agent level part* in Figure 3.2) performed by the respective PLAN WORKERS (cf. Figure 3.3). The agent adopting the role of the PLAN COORDINATOR of a specific plan ρ thereby supervises the control and data flow using the coordination information $\mathcal{CI}^\rho$ encoded for its ensemble level part. Agents within the respective ensemble $\mathcal{E}^\rho$ adopt the role of a PLAN WORKERS that cooperatively work through the plan executing the capabilities according to the cooperation pattern $\mathcal{CP}_t^\rho$ encoded for their agent level parts of the Ensemble Program. PLAN WORKERS send information concerning their current execution state to the PLAN COORDINATOR. They do this for enabling the agent adopting the role of a PLAN COORDINATOR to synchronize control flow and data flow as defined in the $\mathcal{CI}^\rho$ and derive decisions required for making progress in ρ. For determining when to synchronize their state with the PLAN COORDINATOR and other PLAN WORKERS, they follow the commands contained in their respective agent level parts $\mathcal{CP}_t^\rho$. As the $\mathcal{CP}_t^\rho$ and $\mathcal{CI}^\rho$ encode the instructions initially designed by the user in the HTN the plan ρ originates from, with this procedure we can ensure the ensemble cooperatively working on ρ produces the result as desired and as defined by the Ensemble Program.

Running Example: Coordinating the Plan We depict a possible work-flow for coordinating and working on a plan ρ by continuing our running example in Section 3.2.7 after illustrating the functionality and responsibilities of the AGENT LAYER including the role of PLAN WORKERS in Figure 3.13. Here, we continue our running example only focusing on the coordinating part performed by α_2 in its role of the PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$ (cf. Figure 3.9). Because α_2 also adopts the role of the only PLAN WORKER IN $\rho_{\text{NBL-S}}$ required by $\rho_{\text{NBL-S}}$, i.e, WORK_1 for the only task t_1, agent α_2 first instructs itself to execute the necessary capabilities encoded in the respective $\mathcal{CP}_{t_1}^{\text{NBL-S}}$. As a reaction, it receives a synchronization message from itself in its role of a PLAN WORKER IN $\rho_{\text{NBL-S}}$, indicating that from the viewpoint of WORK_1 the coordination of the Ensemble Program for $\rho_{\text{NBL-S}}$ can continue with the next step of the plan.

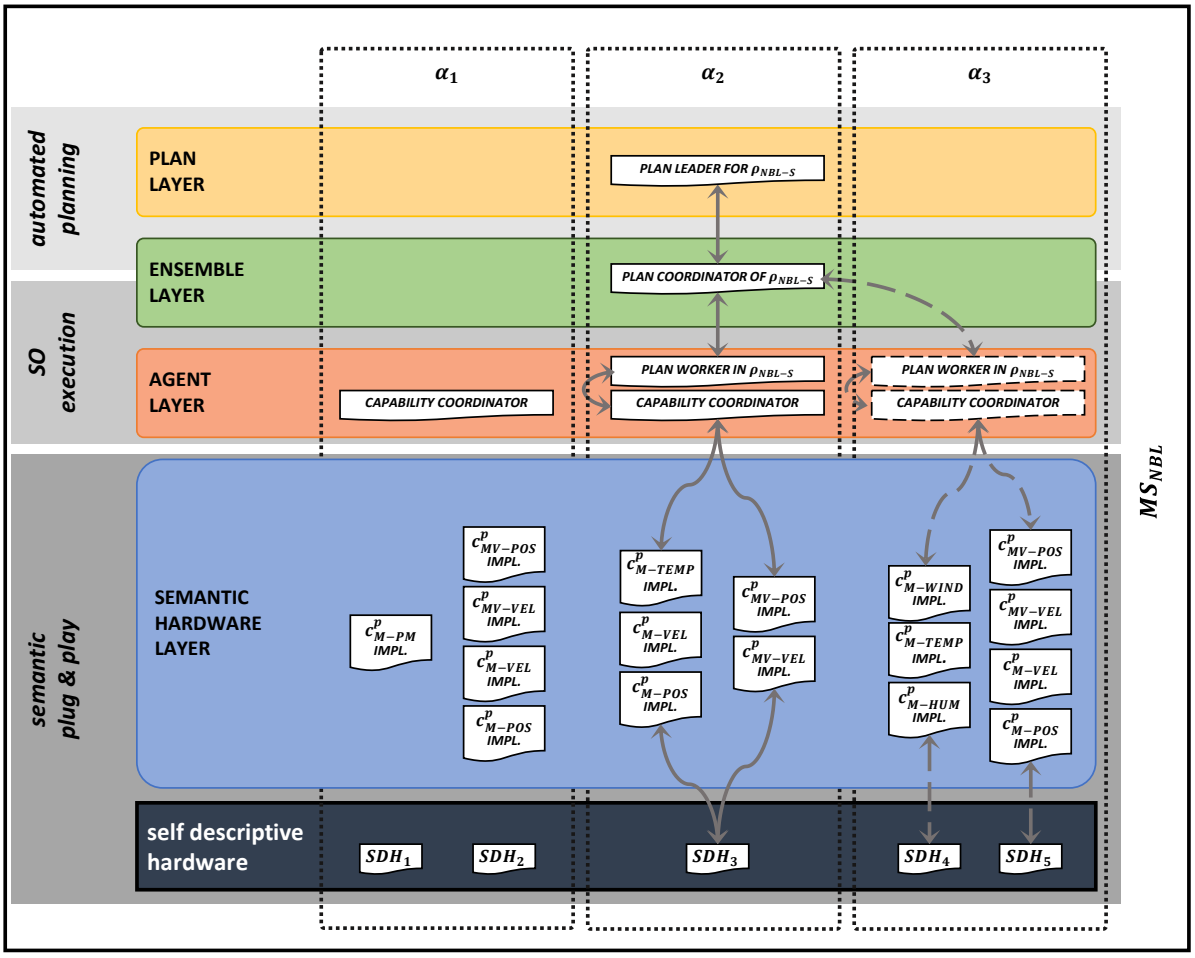


Figure 3.9: Concrete instantiation of a Multipotent System $\mathcal{MS}_{\text{NBL}}$ with agents $\{\alpha_1, \alpha_2, \alpha_3\} \in \mathcal{A}_{\text{NBL}}$ and $\{\text{SDH}_1, \text{SDH}_2, \text{SDH}_3, \text{SDH}_4, \text{SDH}_5\} = \mathcal{SDH}_{\text{NBL}}$ during *working* on one specific plan $\rho_{\text{NBL-S}}$ derived from HTN_{NBL}. Agents α_1, α_2, and α_3 adopt different roles required during working on $\rho_{\text{NBL-S}}$. Agent α_1 is not involved because it cannot provide the needed capability $c^p_{\text{M-TEMP}}$. Because $\rho_{\text{NBL-S}}$ requires only one agent adopting the role of a PLAN WORKER IN $\rho_{\text{NBL-S}}$, only agent α_2 adopts this role while agent α_3 represents an alternative (dashed arrows and lines). Thus, only α_2 actively performs its role of a CAPABILITY COORDINATOR for working on $\rho_{\text{NBL-S}}$. If $\rho_{\text{NBL-S}}$ would require two agents, both α_2 and α_3 would become PLAN WORKER IN $\rho_{\text{NBL-S}}$ and form an ensemble together to be coordinated by α_2. On SEMANTIC HARDWARE LAYER, all agents $\alpha \in \mathcal{A}_{\text{NBL}}$ adopt the roles of c IMPLEMENTER for every capability c they can provide according to their set of SDH they currently are configured with. For working on $\rho_{\text{NBL-S}}$, agent α_2 thus communicates in its role of a CAPABILITY COORDINATOR on AGENT LAYER with itself addressing the relevant roles on SEMANTIC HARDWARE LAYER.

From its $\mathcal{CI}^{\text{NBL-S}}$, agent α_2 knows which PLAN WORKER IN $\rho_{\text{NBL-S}}$ are involved in working on $\rho_{\text{NBL-S}}$ in the ensemble it coordinates in the role of the PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$. Because in our running example, there are no other PLAN WORKER IN $\rho_{\text{NBL-S}}$ than WORK_1, α_2 can continue in coordinating the ensemble working on $\rho_{\text{NBL-S}}$. If there were other PLAN WORKER IN $\rho_{\text{NBL-S}}$, i.e., other agents $\alpha_{i\neq 2} \in \mathcal{A}_{\text{NBL}}$ agent α_2 would need to also receive synchronization messages from those agents before continuing. This would be the case when executing alternative (b) using a swarm algorithm instead of alternative (b) (cf. Figure 3.7) or if not only one but two agents should execute the commands from S1. After receiving all synchronization messages, α_2 in the role of the PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$ thus can continue coordinating S2. Therefore, again agent α_2 instructs WORK_1, i.e., itself in the role of a PLAN WORKER IN $\rho_{\text{NBL-S}}$, to execute the necessary capabilities of S2. Again, agent α_2 receives synchronization information from WORK_1 in its role of a PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$ when it has finished executing these capabilities, this time also including information on the resulting data. In our example, this data contains information concerning the temperature measurements performed by PLAN WORKER IN $\rho_{\text{NBL-S}}$ in S2 that α_2 thus can evaluate in the next step S3 of $\rho_{\text{NBL-S}}$ as encoded in its $\mathcal{CI}^{\text{NBL-S}}$. Dependent on the result of this evaluation, agent α_2 in the role of a PLAN COORDINATOR OF $\rho_{\text{NBL-S}}$ either finds that the ensemble it coordinates determined a temperature inversion at location i (cf. Figure 3.5) and thus can finalize the execution of $\rho_{\text{NBL-S}}$ by informing the PLAN LEADER OF $\rho_{\text{NBL-S}}$ on PLAN LAYER about needed planning on HTN_{NBL}. Otherwise, i.e., there was no inversion detected by the ensemble at all, α_2 can instruct WORK_1 to execute the capabilities encoded in RE6, wait for the synchronization message, and inform the PLAN LEADER OF $\rho_{\text{NBL-S}}$.

3.2.6.4 Coordinating Physical Reconfigurations of the System

In a situation where a specific plan's requirements do not comply with the Multipotent System's configuration, we require the ENSEMBLE LAYER to take on a further responsibility (cf. decision before activity *calculate agent reconfigurations* in Figure 3.2). In such situations, the agent adopting the role of the PLAN LEADER (cf. Figure 3.3) cannot form a sufficiently equipped ensemble able to work on the respective plan ρ it wants to handle in its role of the PLAN COORDINATOR. In this case, we propose to exploit the Multipotent System's property for adapting its configuration, i.e., the physical configuration of agents. A reconfiguration then can modify $\zeta(\mathcal{SDH}_{\mathcal{MS}})$, i.e., all $\mathcal{SDH}_\alpha$ concerning their composition of different independent SDH. This influences all involved agent' set of available capabilities $\mathcal{C}_\alpha$ so that in a new configuration $\zeta(\mathcal{SDH}_{\mathcal{MS}})'$ of the Multipotent System, the previously infeasible plan becomes feasible. On ENSEMBLE LAYER, the problem to solve in that situation can be formulated as an instance of the Multi-Agent Resource Allocation (MARA) problem defined by Chevaleyre et al. [2006]. To deal with that problem, we can use a similar approach to that we use for solving the problem of Ensemble Formation: In the role of a PLAN BIDDER (cf. Figure 3.3), every agent is already aware of the requirements concerning capabilities the respective plan ρ has, encoded in $\mathcal{C}_t$ for every $t \in \mathcal{T}^\rho$. Thus, every agent can elaborate on possibilities for appropriately adapting its physical configuration (cf. activity *propose reconfiguration* in Figure 3.2) and suggest these possibilities to an agent requesting such in the role of a RECONFIGURATION AUCTIONEER (cf. Figure 3.3). Therefore, agents can access a common knowledge base containing information on relevant capabilities in the current SCORe mission. With the information derived from

Table 3.4: Agent configuration in $\mathcal{MS}_{\text{NBL}}$ before adaptation.

$\alpha \in \mathcal{A}_{\text{NBL}}$	$\mathcal{SDH}_{\alpha}^{\text{NBL-CO}}$	$\mathcal{C}_{\alpha}$	$\omega(\mathcal{SDH}_{\alpha}^{\text{NBL-CO}})$
α_1	$\{\text{SDH}_2\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$	$-5kg$
α_2	$\{\text{SDH}_1, \text{SDH}_3\}$	$\{c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-PM}}\}$	$+4kg$
α_3	$\{\text{SDH}_4, \text{SDH}_5\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-WIND}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-HUM}}\}$	$\pm 0kg$

that knowledge base, agents further can estimate the effect of new configurations concerning their set of capabilities $\mathcal{C}_{\alpha}$. The same agent already adopting the role of a PLAN COORDINATOR also adopts the role of the respective RECONFIGURATION AUCTIONEER on ENSEMBLE LAYER (cf. Figure 3.3). We propose this solution because the agent is already aware of the necessary information concerning the plan ρ requiring an adaptation of $\zeta(\mathcal{SDH}_{\mathcal{MS}})$. Thus, the agent is the expert for the requirements of ρ, i.e., the number of tasks $t \in \mathcal{T}^{\rho}$ and the respectively needed capabilities $\mathcal{C}_t$ of each task. According to answers it receives from RECONFIGURATION PROPOSERS (cf. Figure 3.3) as response to a Call for Reconfiguration Proposals (CfRP), the RECONFIGURATION COORDINATOR then can determine how to adapt $\zeta(\mathcal{SDH}_{\mathcal{MS}})$ (cf. the activity *calculate agent reconfigurations* in Figure 3.2). In the role of the RECONFIGURATION COORDINATOR (cf. Figure 3.3) the agent then coordinates how agents that are affected from the reconfiguration, i.e., agents where $\mathcal{SDH}_{\alpha} \neq \mathcal{SDH}_{\alpha}{}'$ with $\mathcal{SDH}_{\alpha} \in \zeta(\mathcal{SDH}_{\mathcal{MS}})$ and $\mathcal{SDH}_{\alpha}{}' \in \zeta(\mathcal{SDH}_{\mathcal{MS}})'$, can perform the adaptation of their configuration $\mathcal{SDH}_{\alpha}$ as RECONFIGURATION IMPLEMENTERS (cf. Figure 3.3). While we propose to rely on interaction with the Multipotent System's user for performing physical reconfigurations of robots (cf. Figure 3.3) because of the complexity of exchanging SDH from an agent's $\mathcal{SDH}_{\alpha}$ [Eymüller et al., 2018], we can also think of a completely autonomous version when this is getting feasible technically, e.g., by integrating another agent offering the capability of *adapting the configuration of other agents* into the process of reconfiguration. Combined with the mechanism for Ensemble Formation, adapting the configuration of the Multipotent System using physical reconfiguration of robots can maintain the operability even when facing versatile requirements within plans. Consequently, when there is no reasonable solution to the problem of Ensemble Formation, we enable the Multipotent System to autonomously reconfigure its physical composition, ensuring that a subsequently restarted Ensemble Formation and the respective processing of the plan is successful. We elaborate on further details and provide a solution integrating this strategy in Chapter 6.

Running Example: Coordinating Adaptations of the Multipotent System for Other Plans To illustrate a situation where an adaptation of the current Multipotent System's configuration $\zeta(\mathcal{MS}_{\text{NBL}})$ concerning the combination of agents and SDHs is required, we look at another plan $\rho_{\text{NBL-RE}}$ generated during the handling of HTN_{NBL}.

We assume a temperature inversion was detected at the position $\text{i} = \langle 48.37, 10.89, 570 \rangle$ by an ensemble $\mathcal{E}^{\text{NBL-S}}$ working on the associated plan $\rho_{\text{NBL-S}}$. Further, weather conditions were evaluated to be critical by another ensemble $\mathcal{E}^{\text{NBL-CO}}$ working on the subsequent plan $\rho_{\text{NBL-CO}}$. Thus, after final planning in $\rho_{\text{NBL-CO}}$ for switching to the React phase of the SCORe mission, we

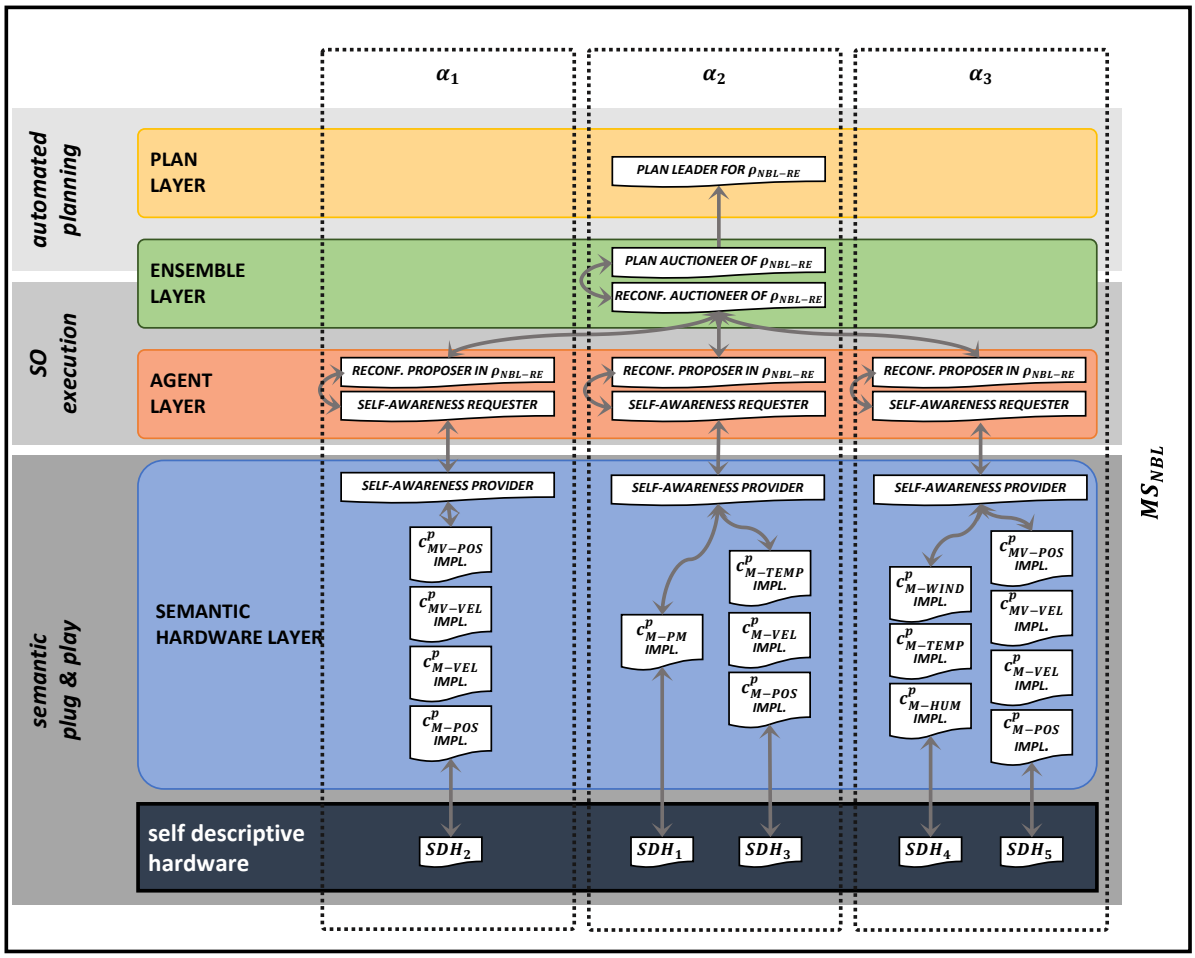


Figure 3.10: Possible concrete instantiation of a Multipotent System $\mathcal{MS}_{\text{NBL}}$ with agents $\{\alpha_1, \alpha_2, \alpha_3\} = \mathcal{A}_{\text{NBL}}$ and $\{\text{SDH}_1, \text{SDH}_2, \text{SDH}_3, \text{SDH}_4, \text{SDH}_5\} = \mathcal{SDH}_{\text{NBL}}$ *during the calculation of a reconfiguration* when facing the plan $\rho_{\text{NBL-RE}}$ derived from HTN_{NBL}. Agent α_2 adopts the role of a RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$ on ENSEMBLE LAYER because it could not find an ensemble $\mathcal{E}^{\text{NBL-RE}}$ in its other role of a PLAN COORDINATOR OF $\rho_{\text{NBL-RE}}$ within $\mathcal{MS}_{\text{NBL}}$. Agents α_1, α_2, and α_3 adopt the roles of RECONFIGURATION PROPOSER IN $\rho_{\text{NBL-RE}}$ on AGENT LAYER when receiving the associated Call for Reconfiguration Proposals (CfRP) initiated by the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$. In their additional roles of SELF-AWARENESS REQUESTERS, α_1, α_2, and α_3 generate information for possible adaptations of their local configurations by communicating with the respective SELF-AWARENESS PROVIDERS whose role they adopt on SEMANTIC HARDWARE LAYER. By answering the request of the SELF-AWARENESS REQUESTER and thereby informing the RECONFIGURATION PROPOSER IN $\rho_{\text{NBL-RE}}$ on AGENT LAYER, agents then can generate a proposal as a response to the CfRP. In this proposal, e.g., α_1 can require a new configuration $\zeta(\mathcal{SDH}_{\text{NBL}}^{\rho_{\text{NBL-CO}}})'$ regarding its set of hardware modules $\mathcal{SDH}_{\alpha_1}$ including, e.g., an SDH encapsulating a S&A for PM measurements.

need our Multipotent System $\mathcal{MS}_{\text{NBL}}$ to work on the respective plan $\rho_{\text{NBL-RE}}$. We further assume that $\mathcal{MS}_{\text{NBL}}$ is configured as depicted in Figure 3.10 after finishing $\rho_{\text{NBL-CO}}$ (cf. Table 3.4).

We know from our analysis in Section 3.2.5.2 that

$$\forall \alpha \in \mathcal{E}^{\text{NBL-RE}} \subseteq \mathcal{A}_{\text{NBL}} : \{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-PM}}\} \subseteq \mathcal{C}_\alpha. \tag{3.1}$$

The only agent providing $c^p_{\text{M-PM}}$ in the assumed configuration is α_2 (cf. Figure 3.10). Unfortunately α_2 cannot provide any capability for movement in its current configuration $\mathcal{SDH}^{\text{NBL-CO}}_{\alpha_2}$ because its total weight of SDH is $\omega(\mathcal{SDH}^{\text{NBL-CO}}_{\alpha_2}) > 0$, indicating it is overloaded. Thus, there is no agent $\alpha \in \mathcal{MS}_{\text{NBL}}$ providing all required capabilities simultaneously. Consequently, we require the Multipotent System $\mathcal{MS}_{\text{NBL}}$ to adapt its configuration appropriately.

Because α_2 adopts the role of the PLAN COORDINATOR OF $\rho_{\text{NBL-RE}}$, it thus requests reconfiguration proposals in a CfRP from all agents within $\mathcal{MS}_{\text{NBL}}$ it can reach in its role of a RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$ (cf. Figure 3.10). In our running example, agents α_1, α_2, and α_3 receive the CfRP and adopt the roles of RECONFIGURATION PROPOSER IN $\rho_{\text{NBL-RE}}$ (cf. Figure 3.10). First, all RECONFIGURATION PROPOSERS IN $\rho_{\text{NBL-RE}}$ analyze their current configurations concerning their respective set of $\mathcal{SDH}^{\text{NBL-CO}}_{\alpha}$ and the resulting set of $\mathcal{C}_\alpha$ in combination to the requirements of tasks $t \in \mathcal{T}^\rho$ of $\rho_{\text{NBL-RE}}$ concerning capabilities $\mathcal{C}_t$. Then, RECONFIGURATION PROPOSERS IN $\rho_{\text{NBL-RE}}$ send their proposals for adapting their local configurations within respective reconfiguration proposals to the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$. In its role of the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$, agent α_2 then can select the best suggestions achieving the required configuration of the Multipotent System from all received proposals. While doing this, the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$ must also ensure the feasibility of realizing the adaptations, e.g., when two or more proposals try to configure the same SDH $\in \mathcal{SDH}_{\text{NBL}}$. For our running example, we assume that α_1 handed in a suggestion $\zeta(\mathcal{SDH}^{\text{NBL-CO}}_{\text{NBL}})'$ for adapting its current set $\mathcal{SDH}^{\text{NBL-CO}}_{\alpha_1}$ by integrating SDH_1 currently integrated with α_2. Thus, this suggestion proposes changing the respective set of available capabilities for agents α_1 and α_2 while α_3 is not affected, i.e.,

$$\mathcal{SDH}^{\text{NBL-CO}'}_{\alpha_1} = \mathcal{SDH}^{\text{NBL-CO}}_{\alpha_1} \cup \text{SDH}_1 = \{\text{SDH}_1, \text{SDH}_2\} \tag{3.2}$$

$$\mathcal{SDH}^{\text{NBL-CO}'}_{\alpha_2} = \mathcal{SDH}^{\text{NBL-CO}}_{\alpha_2} \setminus \text{SDH}_1 = \{\text{SDH}_3\} \tag{3.3}$$

$$\begin{aligned} \zeta(\mathcal{SDH}^{\text{NBL-CO}'}_{\text{NBL}}) &= \{\mathcal{SDH}^{\text{NBL-CO}'}_{\alpha_1}, \mathcal{SDH}^{\text{NBL-CO}'}_{\alpha_2}, \mathcal{SDH}^{\text{NBL-CO}}_{\alpha_3}\} \\ &= \{\{\text{SDH}_1, \text{SDH}_2\}, \{\text{SDH}_3\}, \{\text{SDH}_4, \text{SDH}_5\}\} \end{aligned} \tag{3.4}$$

For actually realizing the physical configuration of agents α_1 and α_2, in its role of a RECONFIGURATION COORDINATOR OF $\rho_{\text{NBL-RE}}$ agent α_2 then instructs α_1 and α_2 in their roles of RECONFIGURATION IMPLEMENTER IN $\rho_{\text{NBL-RE}}$ (cf. Figure 3.11) to interact with the Multipotent System's user for actually modifying the agents' hardware which we focus on during describing the functionality of the AGENT LAYER in Section 3.2.7. After finishing the adaptation of $\mathcal{MS}_{\text{NBL}}$' configuration, agents are then configured following the selected suggestion and providing changed sets of capabilities as depicted in Table 3.5.

Consequently, the subsequently started Ensemble Formation performed by agent α_2 can be completed, allocating the relevant task to α_1 now providing all required capabilities of t_1 in $\rho_{\text{NBL-RE}}$, i.e.,

$$\forall c \in \mathcal{C}_t = \{c^p_{\text{MV-POS}}, c^p_{\text{M-PM}}\} : c \in \mathcal{C}_{\alpha_2}. \tag{3.5}$$

Table 3.5: Configuration of agents in $\mathcal{MS}_{\text{NBL}}$ after the adaptation in the running example.

$\alpha \in \mathcal{A}_{\text{NBL}}$	$\mathcal{SDH}_{\alpha}^{\text{NBL-CO'}}$	$\mathcal{C}_{\alpha}$	$\omega(\mathcal{SDH}_{\alpha}^{\text{NBL-CO'}})$
α_1	$\{\text{SDH}_1, \text{SDH}_2\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-PM}}\}$	$\pm 0kg$
α_2	$\{\text{SDH}_3\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-TEMP}}\}$	$-1kg$
α_3	$\{\text{SDH}_4, \text{SDH}_5\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-WIND}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-HUM}}\}$	$\pm 0kg$

While again, we keep the example and the solution we provide in this section as simple as possible for for the sake of comprehensibility, we provide respectively complex examples in Chapter 6 focusing A Self-Organization Mechanism for Physical Reconfiguration.

3.2.6.5 Future Extensions: Any-time Ensemble Formation

To further increase robustness and autonomy in the ensemble, we can also allow for multiple ensembles in parallel working on different plans simultaneously. We can achieve this with horizontal inter-platform communication on the ENSEMBLE LAYER. This enables further possibilities for coordination, e.g., exchanging agents among ensembles, when necessary. We can use this to compensate for hard failures (break down of agents) and soft failures (battery depletion) during working on a plan. This becomes relevant for phases of a SCORe mission that do not inherently provide robustness against failures, i.e., that do not apply swarm behavior but contain classically planned instruction. To achieve this, we can combine a transactional task execution mechanism with an extended coalition formation algorithm that allows for exchanging agents among ensembles. Although we do not focus on possible failures of robots happening offside of swarm algorithms providing robustness inherently, we further elaborate on the necessary concepts and algorithms we can integrate into future work in Section 7.7.

3.2.7 The Agent Layer

In a Multipotent System, the responsibilities of the AGENT LAYER are twofold. First of all, the AGENT LAYER is responsible for working on plans. For achieving this, agents in an ensemble need to cooperate in the role of PLAN WORKERs on AGENT LAYER for a specific plan ρ. Further, they need to synchronize their execution state with the agent adopting the role of a PLAN COORDINATOR of an ensemble $\mathcal{E}^{\rho}$ on ENSEMBLE LAYER (cf. Figure 3.3). Together, this allows agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ to collectively implement the activities *run ensemble level part* and *run agent level part* in Figure 3.2 necessary for executing Ensemble Programs. While the structural coordination (i.e., Ensemble Formation for a plan) and the coordination of the cooperative execution are handled on the ENSEMBLE LAYER, the AGENT LAYER is responsible for actually working on the respective plans. Thus we allow for communication between AGENT LAYER and ENSEMBLE LAYER *vertically* on the same agent (intra-platform), but also between the AGENT LAYERs and ENSEMBLE LAYERs of different agents (inter-platform) *diagonally* (cf. activity *run ensemble level part* and *run agent level part* on agents α_1 and $\{\alpha_1, ..., \alpha_n\}$ in Figure 3.2). This enables exchanging necessary information between cooperating agents for Ensemble Formation

and calculating new physical configurations for $\mathcal{MS}$, but also for their cooperation during executing plans, e.g., exchanging measurements during the executing of Collective Capabilities encapsulating swarm behavior like, e.g., PSO.

The second responsibility of the AGENT LAYER is to provide information on an agent's current set of capabilities $\mathcal{C}_\alpha$ to other agents in $\mathcal{MS}$ when required. Therefore, the AGENT LAYER interacts with the SEMANTIC HARDWARE LAYER to generate up-to-date information using a self-awareness mechanism. This enables the agent, e.g., to evaluate its competence in working on a plan by analyzing the required capabilities of a task $t \in \mathcal{T}^\rho$. Thereby, the AGENT LAYER plays the relay between the SEMANTIC HARDWARE LAYER of the same agent holding local information and the ENSEMBLE LAYER of other agents aggregating this information. We use this, e.g., to let the agents autonomously detect situations that require the agent or even the whole ensemble to be reconfigured, handle such situations by proposing beneficial exchanges of SDH, and perform adaptations of physical configuration if required.

3.2.7.1 Using Self-Awareness for Participating in Ensemble Formation

Self-awareness on AGENT LAYER is necessary for determining whether an agent fulfills the requirements of a plan ρ in its current hardware configuration $\mathcal{SDH}_\alpha$ determining the set of capabilities $\mathcal{C}_\alpha$ it can provide and execute. Getting aware of this becomes relevant when the agent receives a CfP sent by a PLAN AUCTIONEER (cf. activity *participate if executable* in Figure 3.2). Subsequently, the agent then adopts the role of a PLAN BIDDER (cf. Figure 3.3) and starts evaluating its qualification for participating in the ensemble $\mathcal{E}^\rho$ dedicated to working on the respective plan ρ, concerning its set of capabilities $\mathcal{C}_\alpha$. Thereby, the agent aims at generating a proposal it can send to the PLAN AUCTIONEER indicating in which way it can participate in ρ precisely, i.e., which tasks $t \in \mathcal{T}^\rho$ it can work on. Therefore, the respective AGENT LAYER and the SEMANTIC HARDWARE LAYER of the same agent work together: Agents adopt the roles of a SELF-AWARENESS REQUESTER on AGENT LAYER and the role of a SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER (cf. Figure 3.3). While the SELF-AWARENESS REQUESTER receives the command of the PLAN BIDDER to determine the agent's qualification for working on plan ρ (cf. the activity *determine cap availability* in Figure 3.2), it separates requests for the different tasks $t \in \mathcal{T}^\rho$. It requests information concerning the respective requirements individually from the SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER. This is necessary because information concerning the set of capabilities $\mathcal{C}_\alpha$ can change over time, i.e., if the physical composition concerning its set $\mathcal{SDH}_\alpha$ gets modified by the Multipotent System's user during a physical reconfiguration. Thus, SELF-AWARENESS REQUESTER and SELF-AWARENESS PROVIDER need to take care of possible changes in the physical hardware composition of the agent. That way, the agent can determine the consequences for its set of available capabilities $\mathcal{C}_\alpha$ and react appropriately when participating in the process of Ensemble Formation in the role of a TASK BIDDER (cf. Figure 3.3). To determine these consequences, in its role of the SELF-AWARENESS PROVIDER the agent can check whether it adopts every required role of a CAPABILITY IMPLEMENTER (cf. Figure 3.3) requested in the respective task. For all tasks $t \in \mathcal{T}^\rho$, the SELF-AWARENESS PROVIDER then detects whether each capability c from the set required capabilities $\mathcal{C}_t$ can be provided and executed by a CAPABILITY IMPLEMENTER on SEMANTIC HARDWARE LAYER. Then, the SELF-AWARENESS PROVIDER can inform the PLAN BIDDER about the result. The PLAN BIDDER then generates and send a proposal containing this information back to the PLAN AUCTIONEER as a response to the CfP it initiated. If the respective PLAN AUCTIONEER selects the agent for actually participating in the ensemble $\mathcal{E}^\rho$ working on the

Table 3.6: Set of available capabilities $\mathcal{C}_a$ for agents α_1, α_2, and α_3 when adopting the roles of PLAN BIDDERS in $\rho_{\text{NBL-S}}$ in our running example.

$\alpha \in \mathcal{A}_{\text{NBL}}$	$\mathcal{SDH}_\alpha$	$\mathcal{C}_\alpha$	$\omega(\mathcal{SDH}_\alpha)$
α_1	$\{\text{SDH}_1, \text{SDH}_2\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-PM}}\}$	$\pm 0kg$
α_2	$\{\text{SDH}_3\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-TEMP}}\}$	$-1kg$
α_3	$\{\text{SDH}_4, \text{SDH}_5\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-WIND}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-HUM}}\}$	$\pm 0kg$

plan ρ, the agent also adopts the role of a PLAN IMPLEMENTER (cf. Figure 3.3).

Running Example: Participating in Allocation of a Plan We continue our running example further illustrating the process of participating in the allocation of plan $\rho_{\text{NBL-S}}$ from the perspective of an agent adopting the role of a PLAN BIDDER IN $\rho_{\text{NBL-S}}$ like it is illustrated in Figure 3.8, completing the descriptions in the running example from Section 3.2.6.2. Because α_2 became the leader of plan $\rho_{\text{NBL-S}}$, it also adopts the role of the plan's auctioneer (cf. PLAN AUCTIONEER OF $\rho_{\text{NBL-S}}$ in Figure 3.8) and initiates a CfP concerning the tasks $t \in \mathcal{T}^\rho$ with $\rho = \rho_{\text{NBL-S}}$. Agents α_1, α_2, and α_3 receive this CfP and adopt the roles of PLAN BIDDERS IN $\rho_{\text{NBL-S}}$ on AGENT LAYER. To evaluate whether they can participate in the process, α_1, α_2, and α_3 further adopt the roles of SELF-AWARENESS REQUESTERS on AGENT LAYER that can request information from SELF-AWARENESS PROVIDERS on their respective SEMANTIC HARDWARE LAYER. Relevant information required, therefore concern the set of available capabilities $\mathcal{C}_a$ for all agents α_1, α_2, and α_3, derived from their set of $\mathcal{SDH}_\alpha$ as we depict them in Table 3.6 Because the answer of the SELF-AWARENESS PROVIDER of α_1 returns that it lacks the required capability $c^p_{\text{M-TEMP}}$ necessary for $t_1 \in \mathcal{T}^\rho$ and there are no other tasks in $\mathcal{T}^\rho$, it does not create a proposal in its role of a PLAN BIDDER IN $\rho_{\text{NBL-S}}$. Agents α_2 and α_3 instead provide all required capabilities $c \in \mathcal{C}_{t_1}$, i.e., $c^p_{\text{MV-POS}}$, $c^p_{\text{MV-VEL}}$, $c^p_{\text{M-POS}}$, $c^p_{\text{M-TEMP}}$. Thus both generate proposals for t_1, and send those proposals back to the PLAN AUCTIONEER OF $\rho_{\text{NBL-S}}$ that created the CfP (i.e., α_2 adopting this on ENSEMBLE LAYER, cf. Figure 3.8). The PLAN AUCTIONEER OF $\rho_{\text{NBL-S}}$ thus further coordinates the correct selection of received proposals as we describe it in Section 3.2.6.2.

3.2.7.2 Using Self-Awareness for Enabling Physical Reconfiguration

In combination with the SEMANTIC HARDWARE LAYER, an agent can use its self-awareness abilities to propose possible changes to its physical configuration on AGENT LAYER in the role of a RECONFIGURATION PROPOSER (cf. the activity *propose reconfiguration* in Figure 3.2) if this is requested by an agent adopting the role of a RECONFIGURATION AUCTIONEER on ENSEMBLE LAYER (cf. Figure 3.3). The calculation of proposals for reconfiguration becomes necessary if the agent adopting the role of a PLAN COORDINATOR for a specific plan ρ on the ENSEMBLE LAYER (cf. Figure 3.3) does not find a sufficient number of agents capable of working on ρ. As an answer to the subsequently initiated CfRP performed by the PLAN AUCTIONEER (cf. Section 3.2.6.4), each agent can help to handle the situation by calculating a possible

Table 3.7: Configuration of the agents in $\mathcal{MS}_{\text{NBL}}$ after working on $\rho_{\text{NBL-CO}}$ was feasible.

$\alpha \in \mathcal{A}_{\text{NBL}}$	$\mathcal{SDH}_{\alpha}^{\text{NBL-CO}}$	$\mathcal{C}_{\alpha}$	$\omega(\mathcal{SDH}_{\alpha}^{\text{NBL-CO}})$
α_1	$\{\text{SDH}_2\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$	$-5kg$
α_2	$\{\text{SDH}_1, \text{SDH}_3\}$	$\{c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-PM}}\}$	$+4kg$
α_3	$\{\text{SDH}_4, \text{SDH}_5\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-WIND}}, c^p_{\text{M-TEMP}}, c^p_{\text{M-HUM}}\}$	$\pm 0kg$

new configuration $\zeta(\mathcal{SDH}_{\mathcal{MS}})$ of the Multipotent System in its role of a RECONFIGURATION PROPOSER (cf. Figure 3.3) with suggestions based on its local knowledge. Local knowledge involves the agent's currently attached set $\mathcal{SDH}_{\alpha}$ and knowledge about possible other SDH available in the Multipotent System (cf. the activity *access SDH description* in Figure 3.2), e.g., registered during system setup. However, agents do not necessarily know about the configurations of other agents, i.e., they cannot know which SDH is currently mounted to which other agents in $\mathcal{A}_{\mathcal{MS}}$. Thus, an agent on AGENT LAYER can only propose reliable suggestions for reconfiguration on its own set $\mathcal{SDH}_{\alpha}$. The reconfiguration of the Multipotent System as a whole requires coordination on ENSEMBLE LAYER. Therefore, the proposals for reconfigurations calculated by RECONFIGURATION PROPOSERS are communicated to the agent adopting the role of a RECONFIGURATION AUCTIONEER on ENSEMBLE LAYER. The RECONFIGURATION AUCTIONEER OF ρ then analyzes incoming proposals and decides which suggestions are beneficial for the Multipotent System and help with handling the current plan ρ (cf. the activity *calculate agent reconfigurations* in Figure 3.2). Thereby, the RECONFIGURATION AUCTIONEER can consider inter-dependencies between individual proposals RECONFIGURATION PROPOSER cannot be aware of. This can be the case due to the agents' restricted knowledge concerning the current configuration $\zeta(\mathcal{SDH}_{\mathcal{MS}})$ regarding other agents, e.g., situations where two proposals from different agents require the same SDH be configured to them. Agents whose proposals for reconfiguration are accepted by the RECONFIGURATION AUCTIONEER then get informed by the agent adopting the role of the RECONFIGURATION COORDINATOR (cf. Figure 3.3). To be able to implement the reconfiguration as suggested in their proposal, these agents then adopt the additional role of RECONFIGURATION IMPLEMENTER on AGENT LAYER. In this role and in cooperation with the Multipotent System's user, they can exchange SDH according to the solution previously determined by the RECONFIGURATION AUCTIONEER (cf. the activities *execute agent reconfiguration*, *execute physical reconfiguration*, *recognize new SDH*, *update capability executability*, and *update local rules executability* in Figure 3.2). We further elaborate on the details and provide a technical solution for calculating reconfigurations in Chapter 6.

Running Example: Generating Reconfiguration Proposals for a Plan We continue our running example, further illustrating the process of generating proposals containing suggestions for adaptations of the current configuration of $\mathcal{MS}_{\text{NBL}}$ like it is illustrated in Figure 3.10 and described in the running example in Section 3.2.6.4. Because α_2 became the leader of plan $\rho_{\text{NBL-RE}}$, it adapted the role of the PLAN COORDINATOR OF $\rho_{\text{NBL-RE}}$ in which it failed generating an ensemble $\mathcal{E}^{\text{NBL-RE}}$. Consequently, α_2 also adopts the role of the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$ and initiates a respective CfRP. This triggers all agents $\alpha \in \mathcal{A}_{\text{NBL}}$ re-

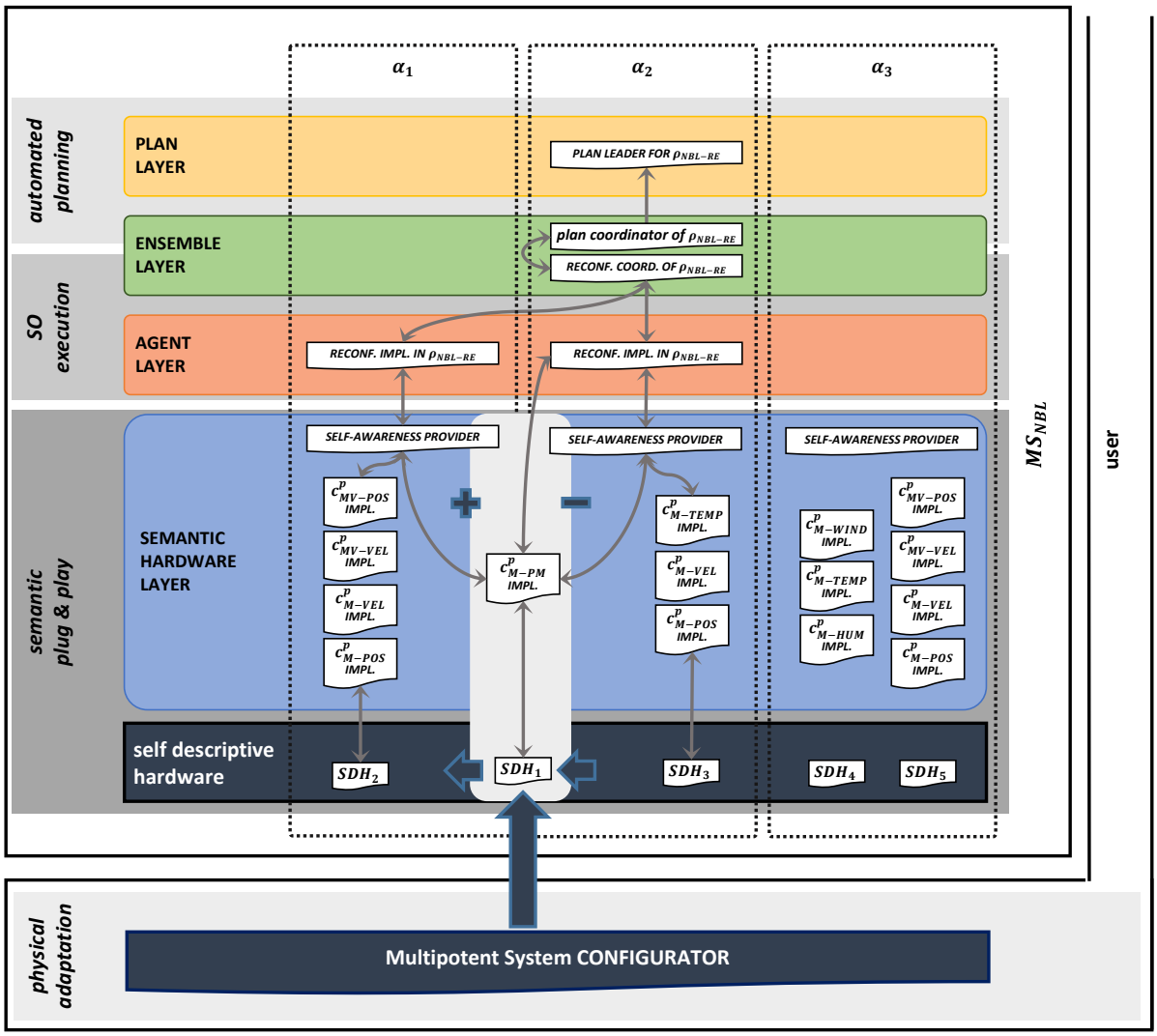


Figure 3.11: Concrete instantiation of a Multipotent System $\mathcal{MS}_{\text{NBL}}$ with agents $\{\alpha_1, \alpha_2, \alpha_3\} = \mathcal{A}_{\text{NBL}}$ and $\{\text{SDH}_1, \text{SDH}_2, \text{SDH}_3, \text{SDH}_4, \text{SDH}_5\} = \mathcal{SDH}_{\text{NBL}}$ *during the execution of a reconfiguration* for enabling $\mathcal{MS}_{\text{NBL}}$ to form an ensemble $\mathcal{E}^{\text{NBL-RE}}$ able to work on the plan $\rho_{\text{NBL-RE}}$. Being the agent adopting the role of the PLAN COORDINATOR OF $\rho_{\text{NBL-RE}}$, agent α_2 also adopts the role of the RECONFIGURATION COORDINATOR OF $\rho_{\text{NBL-RE}}$ for realizing a new configuration $\zeta(\mathcal{SDH}^{\text{NBL-RE}}_{\text{NBL}})$. Agents α_1 and α_2 participate in the reconfiguration and thus adopt the role of a RECONFIGURATION IMPLEMENTER IN $\rho_{\text{NBL-RE}}$ on AGENT LAYER each. Agent α_3 instead keeps its current configuration. To realize the reconfiguration, in its role of a RECONFIGURATION IMPLEMENTER IN $\rho_{\text{NBL-RE}}$, α_2 communicates with the involved $c^p_{\text{M-PM}}$ IMPLEMENTER on SEMANTIC HARDWARE LAYER for initiating the exchange of the associated SDH and indicating the respective information to the user that performs the required physical reconfiguration. As consequence, agent α_1 gains a new capability $c^p_{\text{M-PM}}$ while α_2 looses it. In the then established goal-configuration $\zeta(\mathcal{SDH}^{\text{NBL-RE}}_{\text{NBL}})$, α_1 thus provides all required capabilities c of task t_1 in $\rho_{\text{NBL-RE}}$ and an ensemble $\mathcal{E}^{\text{NBL-RE}}$ can be formed within $\mathcal{MS}_{\text{NBL}}$.

ceiving the CfRP, i.e., α_1, α_2, and α_3, to adopt the role of RECONFIGURATION PROPOSERS IN $\rho_{\text{NBL-RE}}$ and generate reconfiguration proposals regarding their respective set of SDH $\mathcal{SDH}_{\alpha_1}^{\text{NBL-RE}}$, $\mathcal{SDH}_{\alpha_2}^{\text{NBL-RE}}$, and $\mathcal{SDH}_{\alpha_3}^{\text{NBL-RE}}$ suggesting how to adapt the current configuration of $\mathcal{MS}_{\text{NBL}}$ for the plan $\rho_{\text{NBL-RE}}$ from their local perspective.

For the example here, we assume the configuration after successfully finishing $\rho_{\text{NBL-CO}}$ to be described by the set partitioning $\zeta(\mathcal{SDH}_{\mathcal{MS}}^{\text{NBL-CO}})$ regarding the Multipotent System $\mathcal{MS}_{\text{NBL}}$ in its current state, i.e.,

$$\begin{aligned}\zeta(\mathcal{SDH}_{\mathcal{MS}}^{\text{NBL-CO}}) &= \left\{\mathcal{SDH}_{\alpha_1}^{\text{NBL-CO}}, \mathcal{SDH}_{\alpha_2}^{\text{NBL-CO}}, \mathcal{SDH}_{\alpha_3}^{\text{NBL-CO}}\right\} \\ &= \left\{\{\text{SDH}_2\}, \{\text{SDH}_1, \text{SDH}_3\}, \{\text{SDH}_4, \text{SDH}_5\}\right\},\end{aligned} \tag{3.6}$$

having agents configured like illustrated in Figure 3.10. In this configuration, the Multipotent System $\mathcal{MS}_{\text{NBL}}$ was able to work on $\rho_{\text{NBL-CO}}$ because agent α_3 with $\mathcal{SDH}_{\alpha_3}^{\text{NBL-CO}} = \{\text{SDH}_4, \text{SDH}_5\}$ provided all required capabilities of the only task t_1 within $\rho_{\text{NBL-CO}}$, i.e.,

$$\forall c \in \mathcal{C}_{t_1} = \{c_{\text{MV-POS}}^p, c_{\text{MV-VEL}}^p, c_{\text{M-WIND}}^p, c_{\text{M-HUM}}^p\} : c \in \mathcal{C}_{\alpha_3}. \tag{3.7}$$

Thus, the agents $\alpha \in \mathcal{A}_{\text{NBL}}$ now are configured with SDH $\in \mathcal{SDH}_{\text{NBL}}$, resulting in the respective set of capabilities $\mathcal{C}_\alpha$ for the different agents we depict in Table 3.7. As we have seen before, to form an appropriate ensemble $\mathcal{E}^{\text{NBL-RE}}$, agents adopting the roles of PLAN WORKER IN $\rho_{\text{NBL-RE}}$ must fulfill the requirements concerning capabilities defined by tasks $t \in \mathcal{T}^\rho$ of $\rho_{\text{NBL-RE}}$. Being an UAV combined with an integrated temperature sensor, SDH_3 normally would provide the set of capabilities $\{c_{\text{MV-POS}}^p, c_{\text{MV-VEL}}^p, c_{\text{M-POS}}^p, c_{\text{M-VEL}}^p, c_{\text{M-TEMP}}^p\}$ to agent α_2 (cf. Table 3.2). Because in addition to SDH_3 also SDH_1 is included in $\mathcal{SDH}_{\alpha_2}^{\text{NBL-CO}}$ which results in a total weight of $+4kg$, i.e.,

$$\omega(\mathcal{SDH}_{\alpha_2}^{\text{NBL-CO}}) = \text{SDH}_3.\text{WEIGHT} + \text{SDH}_1.\text{WEIGHT} = (-1kg) + (+5gk) = +4kg, \tag{3.8}$$

agent α_2 unfortunately looses its capabilities for moving, i.e., can no longer execute $c_{\text{MV-POS}}^p$ and $c_{\text{MV-VEL}}^p$. At the same time α_2 is the only agent $\alpha \in \mathcal{A}_{\text{NBL}}$ offering the capability $c_{\text{M-PM}}^p$ required in t_1 from $\rho_{\text{NBL-RE}}$, i.e., $c_{\text{M-PM}}^p \in \mathcal{C}_t$.

Reasoning on these individual local conditions concerning their set $\mathcal{SDH}_\alpha$ and their resulting sets of capabilities $\mathcal{C}_\alpha$, the agents send proposals suggesting necessary adaptations qualifying them to work on $t_1 \in \mathcal{T}^\rho$ of $\rho_{\text{NBL-RE}}$. We depict such suggestions in Table 3.8.

Agent α_1 thus proposes to extend its current set $\mathcal{SDH}_{\alpha_1}^{\text{NBL-CO}}$ with SDH_1 allowing it to provide the additional capability $c_{\text{M-PM}}^p$ and thus sufficing the requirements. Agent α_2 instead proposes to exchange SDH_3 with SDH_2, re-enabling its capabilities $c_{\text{MV-POS}}^p$ and $c_{\text{MV-VEL}}^p$ for moving (SDH_2 provides a higher payload than SDH_3). Agent α_3 also requires an exchange of SDH from its current set $\mathcal{SDH}_{\alpha_3}^{\text{NBL-CO}}$, i.e., exchanging the SDH encapsulating the weather sensor with SDH_1.

Agents then send these reconfiguration proposals back to the initiator of the CfRP, i.e., agent α_2 in its role of the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$. If an agent's suggestion concerning its new configuration dedicated for $\rho_{\text{NBL-RE}}$, i.e., $\mathcal{SDH}_\alpha^{\text{NBL-RE}}$, is selected by the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$ or another proposal involves one of the SDH included in an agent's current $\mathcal{SDH}_\alpha^{\text{NBL-CO}}$, agents get informed by agent α_2 in its role of a RECONFIGURATION COORDINATOR OF $\rho_{\text{NBL-RE}}$. As we assume that the RECONFIGURATION COORDINATOR OF $\rho_{\text{NBL-RE}}$ selects the proposal of α_1, agents α_1 and α_2 are involved in the reconfiguration and

Table 3.8: Suggestions of agents for adapting the current system's configuration $\zeta(\mathcal{SDH}^{\text{NBL-RE}}_{\text{NBL}})$ to achieve a new configuration where working on $\rho_{\text{NBL-RE}}$ is feasible from ist local point of view.

$\alpha \in \mathcal{A}_{\text{NBL}}$	adaptation for $\zeta(\mathcal{SDH}^{\text{NBL-RE}}_{\text{NBL}})$	resulting $\mathcal{C}_\alpha$	$\omega(\mathcal{SDH}^{\text{NBL-RE}}_{\alpha})$
α_1	$\mathcal{SDH}^{\text{NBL-CO}}_{\alpha_1} \setminus \{\} \cup \{\text{SDH}_1\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-PM}}\}$	$\pm 0kg$
α_2	$\mathcal{SDH}^{\text{NBL-CO}}_{\alpha_2} \setminus \{\text{SDH}_3\} \cup \{\text{SDH}_2\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-PM}}\}$	$\pm 0kg$
α_3	$\mathcal{SDH}^{\text{NBL-CO}}_{\alpha_3} \setminus \{\text{SDH}_4\} \cup \{\text{SDH}_1\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{M-PM}}\}$	$\pm 0kg$

thus adopt the roles of RECONFIGURATION IMPLEMENTER IN $\rho_{\text{NBL-RE}}$ (cf. Figure 3.11). With the help of the Multipotent System's user in its role of a MULTIPOTENT SYSTEM CONFIGURATION, the physical configurations of α_1 and α_2 then are adapted according to the selected proposals. Because α_2 in its role of the RECONFIGURATION AUCTIONEER OF $\rho_{\text{NBL-RE}}$ previously ensured that the new configuration is feasible (i.e., no SDH is allocated to more than one agent at once) and the agents in $\mathcal{A}_{\text{NBL}}$ fulfill the requirements of $\rho_{\text{NBL-RE}}$ in their new configuration, a subsequently started Ensemble Formation is successful. We remark that for the sake of comprehensibility, in the running example we provide here, the new configuration we calculate as the solution is very straightforward. In Chapter 6 we investigate the relevant details of more complicated cases involving a much broader set of possible SDH and higher numbers of agents.

3.2.7.3 Participating in Ensembles by Executing Agent Level Parts

To increase flexibility, allow for generalization, and fit the concepts of Collective Capabilities encapsulating swarm behavior defined on the ENSEMBLE LAYER, we propose that agent level parts be parametrized similarly. Parameters further describe the requirements an agent needs to fulfill for executing the respective agent level part. For example, an agent level part implementing the search for a gradient of a particular measured value can be used for various measurable quantities with a continuous distribution, e.g., a gas's concentration or the temperature level. Depending on its physical configuration (i.e., its hardware composition $\mathcal{SDH}_\alpha$), an agent then has the needed capabilities to search for the gradient of a specific quantity in a specific task $t \in \mathcal{T}^\rho$ from a plan ρ (i.e., $\mathcal{C}_t \in \mathcal{C}_\alpha$). This, in turn, qualifies the agent for participating in an Ensemble Program (i.e., be a part of the $\mathcal{E}^\rho$ for the respective plan ρ) by allocating the respective task $t \in \mathcal{T}^\rho$ (cf. the activity *participate if executable* in Figure 3.2). When agents finally are selected to participate in an ensemble for a specific plan ρ by the respective PLAN COORDINATOR (cf. Figure 3.3), the selection mechanism has ensured that they provide all capabilities $c \in \mathcal{C}_t$ for the respective task $t \in \mathcal{T}^\rho$ and also can execute them. The respective cooperation patterns $\mathcal{CP}^\rho_t$ of the task t they allocate can contain simple commands for executing single capabilities provided by the agent or contain more complex patterns, interweaving the execution of multiple capabilities and communicating with other agents, e.g., for participating in Collective Capabilities encapsulating swarm behavior. In all cases, we express the specific rule-set encoding the order of execution for these capabilities in $\mathcal{CP}^\rho_t$. For the execution itself,

the cooperation patterns $\mathcal{CP}_t^\rho$ define the communication and data-flow on the inter-agent and intra-agent scope:

1. The $\mathcal{CP}_t^\rho$ defines the **inter-agent** communication within the respective ensemble $\mathcal{E}^\rho$
   a) concerning PLAN WORKER on AGENT LAYER and the PLAN COORDINATOR on ENSEMBLE LAYER, often adopted from different agents $\alpha_{i\neq j} \in \mathcal{E}^\rho$ (cf. Figure 3.3). Communication between different agents here is necessary to coordinate plans in general (cf. Figure 3.2). Therefore, the agent adopting the role of the PLAN COORDINATOR (cf. agent α_1 in Figure 3.2) communicates with all participating agents that adopt the role of PLAN WORKER (cf. agents $\alpha_{1,\ldots,n}$ in Figure 3.2). Together, the PLAN COORDINATOR and the agents adopting the roles of PLAN WORKER thus implement the activities *run ensemble level part*, *run agent level part*s, and *synchronize execution state* from Figure 3.2. Messages transferred this way contain information, e.g., about the starting, the ending, and the results of the agent level parts' execution. Further, we can also use them to synchronize the ensemble during a specific Ensemble Program's execution. During collective transport, e.g., required in the case study on *Innovative Measurement Methods* in Section 2.4, synchronization of the agents' movement in the ensemble is urgently required. We further elaborate on the details required for such kind of coordination of execution in an ensemble $\mathcal{E}^\rho$ working on plan ρ in Chapter 7 and give an evaluation of the concepts using real hardware in Section 3.4.2 and using simulated hardware in Section 7.6.
   b) concerning the PLAN WORKER on AGENT LAYERS of different agents $\alpha_{i\neq j} \in \mathcal{E}^\rho$. Communication here is necessary, especially for realizing Collective Capabilities encapsulating swarm behavior. For participating in Collective Capabilities encoding, e.g., cooperative search achieved by an adapted PSO algorithm, a formation flight, or (guided-) flocking, the capabilities we need agents to execute for achieving the desired effect are arranged in the required execution order in a $\mathcal{CP}_t^\rho$ of the respective agent level part generated explicitly for the individual swarm behavior. In the case of the PSO, e.g., it is indispensable for the efficiency of the algorithm that agents can communicate their current measurements to other agents in their ensemble (cf. Figure 3.12). We further elaborate on the execution of Collective Capabilities in Chapter 7.
2. The $\mathcal{CP}_t^\rho$ defines the **intra-agent** communication within the same agent for coordinating the order of execution concerning the relevant capabilities contained in $\mathcal{CP}_t^\rho$. Coordination here is necessary for each agent level part for ensuring the correct order of execution, the starting, the finishing, and the synchronization defined in $\mathcal{CP}_t^\rho$. Concerning the specific execution properties of the capabilities addressed in $\mathcal{CP}_t^\rho$, the agent needs to coordinate the capabilities' execution in its role of a CAPABILITY COORDINATOR. The necessity of the coordination of a capability's execution arises, e.g., from its property defining how many SDH need to be addressed for its execution (cf. the concept of virtual capabilities in Section 3.2.8). Conditions how capabilities can be executed can change during run-time, e.g., after a physical reconfiguration of the Multipotent System. Thus, a capability can be directly executable using only one SDH or may require addressing multiple SDH $\in \mathcal{SDH}_\alpha$ therefore. Depending on its specific configuration concerning $\mathcal{SDH}_\alpha$, for generating a localized measurement of parameter x an agent may require combining the execution of $c^p_{\text{M-POS}}$ for measuring its current position with the execution of the respective capability for measuring the parameter x if the agent does not provide an SDH

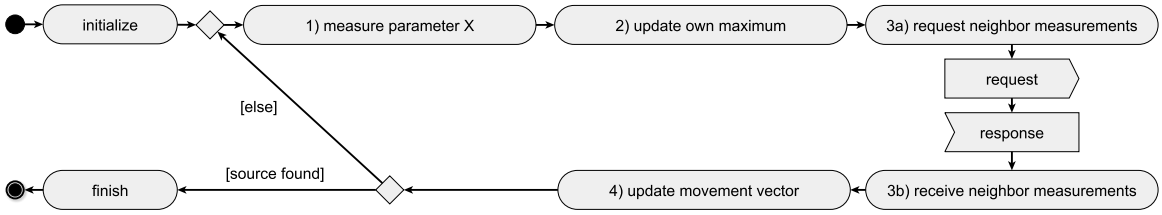


Figure 3.12: Sketch of local rules encapsulated in a $\mathcal{CP}_t^\rho$ of an agent level part necessary for executing the particle swarm optimization algorithm [Zhang et al., 2015]. Each participant follows four simple rules: *1)* Measure the parameter of interest (X), including the location of the measurement. *2)* Update the position of the locally found maximum concentration of X, if necessary. *3)* Request *a)* and receive *b)* measurements of all neighbors for determining the local maximum of X and wait for responses. *4)* Adjust the current moving direction according to the weighted average of the local maximum of X, the global maximum of X, and a random component (weights can be adjusted over time).

Table 3.9: Definition of the virtual capability $c^v_{\text{TEMP-GRAD}}$ from the running example.

capability	description	input value	return value
$c^v_{\text{TEMP-GRAD}} := \{c^p_{\text{M-TEMP}} \bigstar c^p_{\text{MV-VEL}}\}$	ascending while temperature gradient slope stays uniform	maximum height h	$\langle$ LAT, LON, ALT $\rangle$

directly providing the capability alone. Further, capabilities differ in the way they generate a callback to the CAPABILITY COORDINATOR commanding their execution. On the one hand, there are capabilities automatically generating a callback after finishing their execution, e.g., a capability for measuring parameter x typically returns the measured value included in its callback as soon as the respective SDH derives the measurement. On the other hand, some capabilities cannot terminate their execution independently, e.g., moving with a given velocity cannot return a callback indicating that the execution of the respective capability has finished. Instead, it can only indicate that its execution has successfully been started on the respective SDH and requires further coordination to terminate its execution. For both termination types of capabilities and all possible combinations thereof defined in $\mathcal{CP}_t^\rho$, an agent needs to coordinate the execution of the respective capabilities in its role of a CAPABILITY COORDINATOR (cf. Figure 3.3) to correctly implement the activities *execute capabilities*, STOP CAPABILITIES' EXECUTION, and *register execution finished* on the subordinated SEMANTIC HARDWARE LAYER. We further elaborate on the technical details for executing capabilities using SDH and combinations thereof in Chapter 7.

Running Example: Working on Plans Cooperatively For our running example, we assume a slightly modified version of the plan $\rho_{\text{NBL-S}}$ in this chapter, i.e., $\rho_{\text{NBL-S}'}$, requiring two

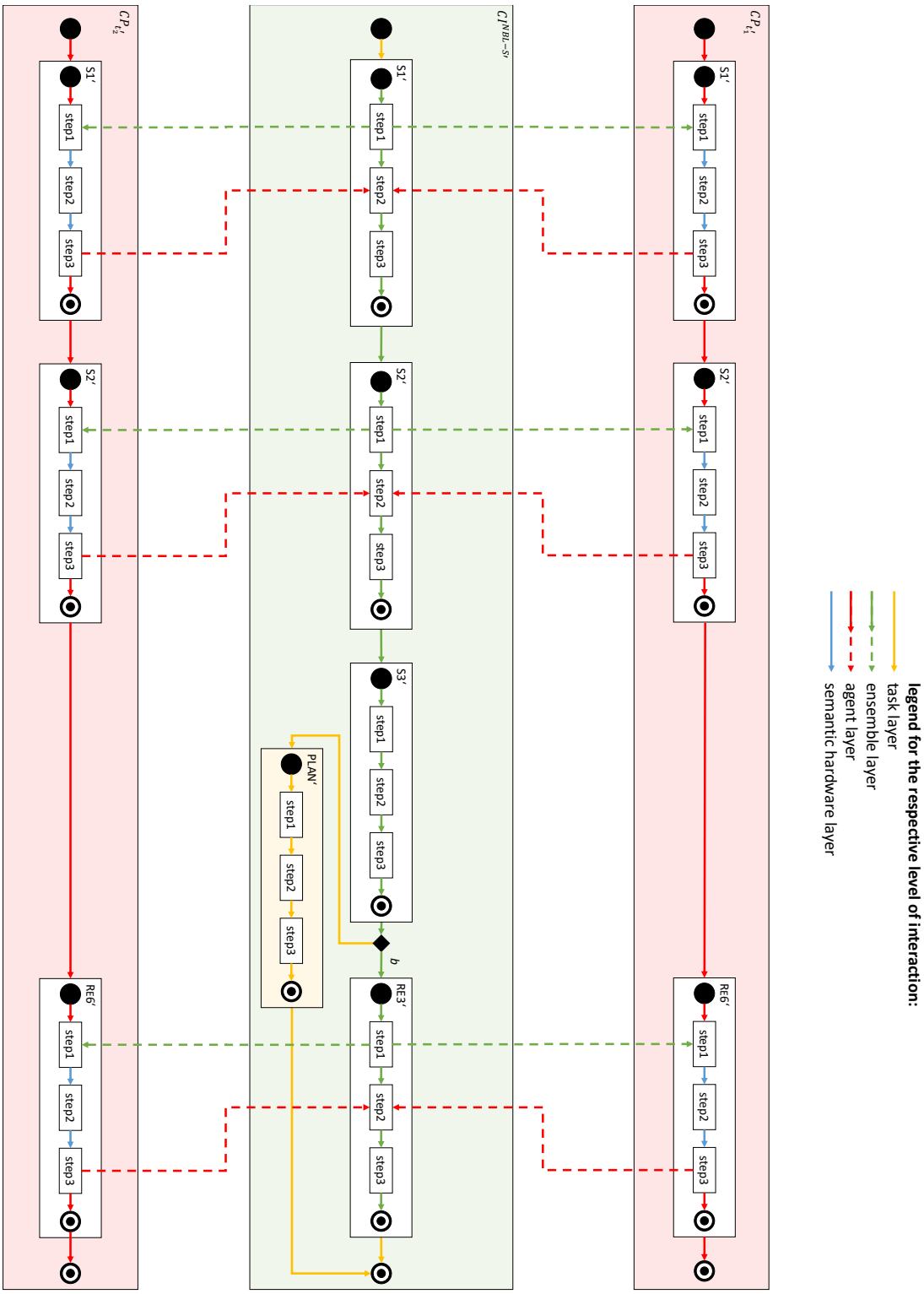
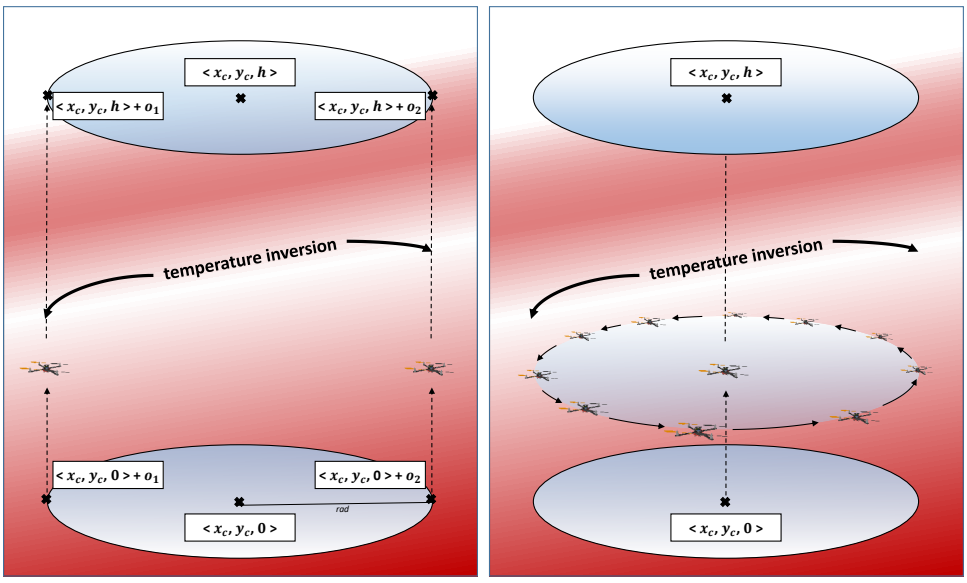


Figure 3.13: Work-flow in $\rho_{\text{NBL-S}'}$ illustrating the communication between COORD′ on ENSEMBLE LAYER (green lane in the middle) and WORK′$_1$ and WORK′$_2$ on AGENT LAYER (red lanes on the bottom and the top). Colors of directed edges between activities indicate which layer is addressed: Yellow edges address PLAN LAYER, green edges address ENSEMBLE LAYER, red edges address AGENT LAYER, and blue edges address SEMANTIC HARDWARE LAYER. We depict intra-agent communication with bold edges and inter-agent communication with dashed edges.



(a) two agents (b) Collective Capability *Ring-of-Fliers*

Figure 3.14: Part S2' of $\rho_{\text{NBL-S}'}$ using an ensemble $\mathcal{E}^{\text{NBL-S}'}$ requiring more than one agent adopting the roles of PLAN WORKER IN $\rho_{\text{NBL-S}'}$ for performing a measurement flight for detecting a temperature inversion.

agents as PLAN WORKER IN $\rho_{\text{NBL-S}'}$ for two different tasks t'_1 and t'_2 instead of only one task t_1 like in the original plan $\rho_{\text{NBL-S}}$ we handle in Section 3.2.6. Thus, we require a PLAN WORKER *Work*$'_1$ and a PLAN WORKER *Work*$'_2$ to work cooperatively on $\rho_{\text{NBL-S}'}$ (cf. Figure 3.13). We further assume that we successfully have allocated the respective tasks to agents adopting these roles. Agent α_2 as Work$'_1$ and agent α_3 as Work$'_2$ form an ensemble $\mathcal{E}^{\text{NBL-S}'}$ that is coordinated by α_2 in its role of the PLAN COORDINATOR OF $\rho_{\text{NBL-S}'}$ (*Coord*$'$). We know this is feasible because we had an alternative allocation of α_3 available during the handling of $\rho_{\text{NBL-S}}$ (cf. Figure 3.9). We now can use this alternative as an additional allocation in $\rho_{\text{NBL-S}'}$. Obviously, if we had a larger ensemble available consisting of n agents providing the required capabilities $c^p_{\text{M-TEMP}}$, $c^p_{\text{MV-POS}}$, and $c^p_{\text{MV-VEL}}$, we could extend $\rho_{\text{NBL-S}'}$ with a respective amount of tasks $t_{3,\ldots,n}$ for Work$_{3,\ldots,n}$.

Similar to the ensemble $\mathcal{E}^{\rho\rho_{\text{NBL-S}}}$ for the plan $\rho_{\text{NBL-S}}$, we want our ensemble $\mathcal{E}^{\text{NBL-S}'}$ in $\rho_{\text{NBL-S}'}$ to perform a coordinated measurement flight with a synchronization after completing S1$'$ (cf. Figure 3.5) for ensuring time synchronized measurements afterward. To improve measurements with a higher spatial variability, we can use different geographic coordinates as parameter for $c^p_{\text{MV-POS}}$ in S1$'$ in every task of $\rho_{\text{NBL-S}'}$ defined by offsets o_1 and o_2 from the original position c with $c := \langle \text{LAT}_c, \text{LON}_c, \text{ALT}_c \rangle$. For calculating them, we define a circle in the plane parallel to the ground surface with a radius of *rad* around the center $\langle \text{LAT}_c, \text{LON}_c, \text{ALT}_c \rangle$ and use equally distributed positions on the circular line we define that way (cf. Figure 3.14a).

Further in S2$'$, we want to achieve that the whole ensemble $\mathcal{E}^{\text{NBL-S}'}$ stops its execution after all agents in the ensemble detected the temperature inversion. Thus, we require Work$'_1$ and Work$'_2$ to detect a temperature inversion online (cf. Figure 3.14). We assume to have a

respective capability $c^v_{\text{TEMP-GRAD}}$ available for agents α_2 and α_3, enabling them to move with a given velocity until they detect a change in the dominant temperature gradient (i.e., a change of the slope of values from temperature measurements from *decreasing* to *increasing* within a given sliding window of measurements in the case of a temperature inversion, cf. Section 2.3). We depict the definition of $c^v_{\text{TEMP-GRAD}}$ in Table 3.9. We are allowed to make this assumption, because we can realize such virtual capability (cf. Section 3.2.8) by appropriately combining the two physical capabilities $c^p_{\text{M-TEMP}}$ and $c^p_{\text{MV-VEL}}$ that agents α_2 and α_3 both provide on SEMANTIC HARDWARE LAYER to a virtual capability $c^v_{\text{TEMP-GRAD}} := \{c^p_{\text{M-TEMP}} \bigstar c^p_{\text{MV-VEL}}\}$ (cf. Figure 3.15 and running example of Section 3.2.8). This means, if an agent provides $c^p_{\text{M-TEMP}}$ and $c^p_{\text{MV-VEL}}$, it can also provide the virtual capability $c^v_{\text{TEMP-GRAD}}$.

To make progress in $\rho_{\text{NBL-S}'}$, we then can use the final positions of WORK'_1 and WORK'_2, i.e., WORK'_1.POS.ALT and WORK'_2.POS.ALT, for determining whether to progress with PLAN or RE6, i.e., whether the phenomena of a temperature inversion was detected by the agents in the ensemble $\mathcal{E}^{\text{NBL-S}'}$ below the defined height h (cf. Figure 3.5). In the case, $\mathcal{E}^{\text{NBL-S}'}$ detected a temperature inversion, we can use the geographic coordinates representing the centroid of all agents $\alpha \in \mathcal{E}^{\text{NBL-S}'}$ as a goal destination for the follow-up plan $\rho_{\text{NBL-CO}}$ (cf. Figure 3.5). Thus we need to let $\rho_{\text{NBL-S}'}$ calculate the respective position and store it to the world state WS in $\text{S3}'$, making the data available for subsequent planning on HTN_{NBL}.

For realizing such a plan, we require to generate appropriate cooperation patterns $\mathcal{CP}^{\text{NBL-S'}}_{t'_1}$ for WORK'_1 (i.e., for agent α_2 in its role of a PLAN WORKER IN $\rho_{\text{NBL-S}'}$) and $\mathcal{CP}^{\text{NBL-S'}}_{t'_2}$ for WORK'_2 (i.e., for agent α_3 in its role of a PLAN WORKER IN $\rho_{\text{NBL-S}'}$) defining when to exchange information with agent COORD' (i.e., α_2 in its role of the PLAN COORDINATOR OF $\rho_{\text{NBL-S}'}$). Complementary to the $\mathcal{CP}^{\rho}_{t}$ for PLAN WORKERS, we require a respective coordination pattern $\mathcal{CI}^{\text{NBL-S}'}$ for COORD' (i.e., for agent α_2 in its role of a PLAN COORDINATOR OF $\rho_{\text{NBL-S}'}$) encoding the necessary steps for coordinating the ensemble, i.e., all PLAN WORKERS involved. Because we require all agents $\alpha \in \mathcal{E}^{\text{NBL-S}'}$ adopting the role of PLAN WORKER ON $\rho_{\text{NBL-S}'}$ to perform very similar actions in our running example, the cooperation pattern $\mathcal{CP}^{\text{NBL-S'}}_{t'_1}$ for WORK'_1 (i.e., agent α_2) and the cooperation pattern $\mathcal{CP}^{\text{NBL-S'}}_{t'_2}$ for WORK'_2 (i.e., agent α_3) in $\rho_{\text{NBL-S}'}$ only slightly differs concerning the offset o_2 in $\mathcal{CP}^{\textit{NBL-S'}}_{t'_2}$.*step2* and the result sent in $\text{S2}'$ (cf. Table 3.10). The coordination information $\mathcal{CI}^{\text{NBL-S}'}$ relevant for the COORD' (agent α_2) contains the information necessary for coordination we depict in Table 3.12. Then, the ensemble $\mathcal{E}^{\text{NBL-S}'}$ can start working on $\rho_{\text{NBL-S}'}$ by instructing COORD' to send an initial coordination signal to WORK'_1 and WORK'_2, indicating that they can start executing their agent level parts with the first instructions encoded in their respective cooperation patterns (i.e., those instructions encoded in $\text{S1}'$). Because the execution of capability $c^p_{\text{MV-POS}}$ in $\text{S1}'$ terminates by itself when the respective agent reaches the position, we can integrate a command for sending a synchronization message to the COORD' in the respective cooperation pattern ($\mathcal{CP}^{\text{NBL-S'}}_{t'_1}$, or $\mathcal{CP}^{\text{NBL-S'}}_{t'_2}$) within the associated agent level part. After receiving all synchronization messages it waits for (i.e., that of WORK'_1 and WORK'_2), COORD' can progress with the coordination of $\rho_{\text{NBL-S}'}$ as defined in the $\mathcal{CI}$ included in its ensemble level part. Which PLAN WORKERS need to synchronize their execution for making progress in the ensemble level part is defined in the $\mathcal{CI}^{\text{NBL-S}'}$ generated for $\rho_{\text{NBL-S}'}$ (there might be such, not able to synchronize on their own as they are executing capabilities that cannot terminate on their own, cf. Section 3.2.8).

The ensemble then performs the same procedure for $\text{S2}'$, executing the associated capabil-

ities each as defined in the respective $\mathcal{CP}_{t'}^{\text{NBL-S'}}$. After this execution, COORD′ then requires to receive the positions of all PLAN WORKER IN $\rho_{\text{NBL-S}'}$ (i.e., that of WORK_1' and WORK_2') for deciding on the next node after the measurement flight in S2′. If there is no inversion detected below the defined height h, WORK_1' and WORK_2' reported a final position WORK′.POS with WORK′.POS.ALT $\geq h$. Thus, we want we want all agents $\alpha \in \mathcal{E}^{\text{NBL-S}'}$ to return to the user's position u in RE6 (cf. Figure 3.5). If, otherwise, there is an inversion present, COORD′ requires the final positions of all agents $\alpha \in \rho_{\text{NBL-S}'}$ for calculating the centroid for updating the world state WS correctly enabling a subsequent planning on $\text{HTN}_{\text{NBL}'}$ in PLAN′ to work correctly (cf. Figure 3.5). We further illustrate this interaction between WORK_1', WORK_2', and COORD′ necessary for accomplishing the plan $\rho_{\text{NBL-S}'}$ in Figure 3.13 referencing the respective cooperation pattern from Table 3.10 and the coordination information from Table 3.12.

When integrating more agents into the process of finding an inversion layer, instead of precisely planning actions for individual agents requiring, e.g., calculations of respective positions and movements in S2′ each, we can also rely on applying appropriate swarm behavior encapsulated in a Collective Capability. Like we already stated in the course of our running example in Section 3.2.6, useful behavior in an ensemble concerning this application can be generated by applying the Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior as used in alternative (b) for plan $\rho_{\text{NBL-S}}$, i.e., with an ensemble $\mathcal{E}^{\text{NBL-S(B)}}$ (cf. Figure 3.14b). When doing so, instead of ascending to height h independently at the pre-calculated positions encoded in S2′, we can let the members of the ensemble executing the respective Collective Capability find their positions themselves cooperatively: By communicating their positions to other agents $\alpha \in \mathcal{E}^{\text{NBL-S(B)}}$, agents can find and hold their position on the circular line. Likewise, they can take care of the relative distance to each other, generating a formation holding equally distributed distances on the circular line. Because we can use a dedicated moving target (e.g., a specific user-controlled UAV) whose position agents are aware of (e.g., by letting the target actively communicate it to the agents), we can easily move the whole ensemble towards positions we require it. In Chapter 7, we illustrate how we can use Collective Capabilities similarly to other capabilities in a goal-oriented fashion and how agents in an ensemble can realize the described *Ring-of-Fliers* swarm behavior or other swarm behavior by only changing the parameters we execute a respective Collective Capability with. In our example here, we make use of executing Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior with changing parameters for achieving the respective desired effect: By moving the target to c first without additional parameters, then upward to height h while additionally executing $c^{v}_{\text{TEMP-GRAD}}$, and again without any additional parameter to u subsequently, we can generate a similar behavior like that of $\rho_{\text{NBL-S}}$ (a) without restrictions in the ensemble's scale.

When executing a Collective Capability instead of other capabilities, the cooperation pattern $\mathcal{CP}_{t'}^{\text{NBL-S'}}$ for each agent adopting the role of a WORKER IN $\rho_{\text{NBL-S}'}$ and the coordination information $\mathcal{CI}^{\text{NBL-S'}}$ for COORD′ change respectively. Instead of requiring differently designed cooperation pattern for each agent $\alpha \in \mathcal{E}^{\text{NBL-S}'}$, in case of using Collective Capabilities, we only require one generic $\mathcal{CP}_{\text{SWARM}}^{\text{NBL-S}}$ that is identical for all agents (cf. Table 3.11). This way, we can exploit the beneficial property of scalability provided by the concept of a Collective Capability at plan design time and execution time. Further, we can also exploit the inherent robustness of the Collective Capability if necessary, which we achieve by the cooperation of the other ensemble members: In the (in this example unlikely) event of an individual agent's failure, other agents compensate for this autonomously.

Table 3.10: The cooperation pattern $\mathcal{CP}^{\text{NBL-S}'}_{t'_i}$ for WORK'_i ($\text{WORK}'_1 := \alpha_2$, and $\text{WORK}'_2 := \alpha_3$), extensible for more agents by planning additional $\mathcal{CP}^{\text{NBL-S}'}_{t_i}$.

node	$\mathcal{CP}^{\text{NBL-S}'}_{t'_i}$.step1	$\mathcal{CP}^{\text{NBL-S}'}_{t'_i}$.step2	$\mathcal{CP}^{\text{NBL-S}'}_{t'_i}$.step3
S1′	wait for start signal from COORD′	execute $c^{p}_{\text{MV-POS}}$ to position $\langle \text{LAT}_c, \text{LON}_c, \text{ALT}_c \rangle + o_i$	synchronize with COORD′
S2′	wait for start signal from COORD′	execute $c^{v}_{\text{TEMP-GRAD}}$ to maximum height h	send final position WORK'_i.POS to COORD′
S3′	—	—	—
PLAN′	—	—	—
RE6′	wait for start signal from COORD′	execute $c^{p}_{\text{MV-POS}}$ to position $\langle \text{LAT}_u, \text{LON}_u, \text{ALT}_u \rangle$	synchronize with COORD′

Table 3.11: The cooperation pattern $\mathcal{CP}^{\text{NBL-S}'}_{\text{SWARM}}$ identical for any agent $\alpha \in \mathcal{E}^{\text{NBL-S}'}$ when using a Collective Capability encapsulating a *Ring-of-Fliers* swarm behavior.

node	$\mathcal{CP}^{\text{NBL-S}}_{\text{SWARM}}$.step1	$\mathcal{CP}^{\text{NBL-S}}_{\text{SWARM}}$.step2	$\mathcal{CP}^{\text{NBL-S}}_{\text{SWARM}}$.step3
S1′	wait for start signal from COORD′	execute Collective Capability *Ring-of-Fliers*	synchronize with COORD′
S2′	wait for start signal from COORD′	execute Collective Capability *Ring-of-Fliers* with $c^{v}_{\text{TEMP-GRAD}}$	send final position WORK'_s.POS to COORD′
S3′	—	—	—
PLAN′	—	—	—
RE6′	wait for start signal from COORD′	execute Collective Capability *Ring-of-Fliers*	synchronize with COORD′

3.2.8 The Semantic Hardware Layer

To enable an agent to deduce available capabilities from specific physical SDH configurations and for realizing the actual execution of capabilities using these SDHs, we implement the SEMANTIC HARDWARE LAYER as a software adapter to hardware. The SEMANTIC HARDWARE LAYER, therefore, encapsulates the self-awareness functionality of an agent. Enriched by a common knowledge base that stores relations between SDH and capabilities, the SEMANTIC HARDWARE LAYER manages the set of available capabilities in any possible physical configuration of the agent. By adopting the role of a SELF-AWARENESS PROVIDER the agent provides essential information for the AGENT LAYER to determine its competence of participating in the process of Ensemble Formation initiated by an agent on ENSEMBLE LAYER (cf. communication between SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER and SELF-AWARENESS REQUESTER on AGENT LAYER in Figure 3.3 realizing the activity of *determine cap availability* in Figure 3.2). Further, the agent can use this information to propose adaptations during Physical Reconfiguration concerning SDH in situations where this is needed (cf. communication

Table 3.12: Coordination information $\mathcal{CI}^{\text{NBL-S}'}$ similar independent of whether agents $\alpha \in \mathcal{E}^{\text{NBL-S}'}$ execute a Collective Capability or not (cf. Tables 3.10 and 3.11).

node	$\mathcal{CI}^{\text{NBL-S}'}$.step1	$\mathcal{CI}^{\text{NBL-S}'}$.step2	$\mathcal{CI}^{\text{NBL-S}'}$.step3
S1′	let all WORK′ $\in \mathcal{E}^{\text{NBL-S}'}$ start S1′	wait for synchronization of all WORK′ $\in \mathcal{E}^{\text{NBL-S}'}$	continue with node $n :=$ S2′
S2′	let all WORK′ $\in \mathcal{E}^{\text{NBL-S}'}$ start S2′	wait for synchronization of all WORK′ $\in \mathcal{E}^{\text{NBL-S}'}$	continue with node $n :=$ S3′
S3′	set BOOLEAN $b :=$ $\exists$ WORK′ $\in \mathcal{E}^{\text{NBL-S}}$: WORK′.POS.ALT $< h$,	change world state WS $i := \begin{cases} \text{AVG(WORK′.POS)}, & \text{if } b \\ \text{NULL}, & \text{else} \end{cases}$	continue with node $n := \begin{cases} \text{RE6′}, & \text{if } b \\ \text{PLAN′} & \text{else} \end{cases}$
PLAN′	finalize $\rho_{\text{NBL-S}'}$	trigger planning in $\text{HTN}_{\text{NBL}'}$ (WS was updated in S3′)	finalize plan
RE6′	let all WORK′ $\in \mathcal{E}^{\text{NBL-S}'}$ start RE6′	wait for synchronization of all WORK′ $\in \mathcal{E}^{\text{NBL-S}'}$	finalize plan

between SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER and RECONFIGURATION PROPOSER on AGENT LAYER in Figure 3.3 realizing the activity *propose reconfiguration* in Figure 3.2). When required by the CAPABILITY COORDINATOR on AGENT LAYER, in each role of a CAPABILITY IMPLEMENTER it adopts on SEMANTIC HARDWARE LAYER the agent can execute the respective capability by accessing the relevant SDH. When an CAPABILITY COORDINATOR triggers the execution of a capability in an agent level part, the addressed CAPABILITY IMPLEMENTER forwards this request to the appropriate SDH (cf. EXECUTE CAPABILITY and ACCESS HARDWARE in Figure 3.2), collects up possible results like measurements (cf. STOP CAPABILITY EXECUTION after a STOP EVENT in Figure 3.2), and informs the CAPABILITY COORDINATOR on AGENT LAYER about the execution status (cf. the activity *synchronize execution state* in Figure 3.2). Thus, the SEMANTIC HARDWARE LAYER serves as an adapter between software representation and hardware execution. Thereby, the SEMANTIC HARDWARE LAYER helps us abstracting from concrete hardware implementations of SDH by encapsulating them from their usage on higher layers of the architecture.

As we already stated at the beginning of Section 3.2, the concrete implementation of that layer, including further elaboration on the necessary concepts, is not in the scope of this thesis. Instead, we present the interface to SDH on SEMANTIC HARDWARE LAYER in [Eymüller et al., 2018] and [Wanninger et al., 2018] and analyze further necessary concepts in [Kosak et al., 2018] and [Wanninger et al., 2021]. Nevertheless, we give a short overview here to illustrate the way we can make this abstraction. Here, we focus on the interface between the SEMANTIC HARDWARE LAYER and the AGENT LAYER and the intra-agent communication necessary for the coordinated capability execution and its self-awareness functionality during Ensemble Formation and Physical Reconfiguration.

3.2.8.1 Providing Self-Awareness and Hardware Access

For realizing the self-awareness abilities of an agent, we require to specify further the interface an agent has to the capabilities provided by S&A encapsulated in SDH. We subdivide

the respective description of each SDH into capabilities and properties of the respectively encapsulated S&A. Properties are pieces of static information further describing an SDH, e.g., its geometric dimensions, weight, and measuring units. Moreover, we can use properties to describe further the capabilities provided by an SDH, e.g., when we require a specific precision in measuring. Capabilities are executable actions of an SDH with direct access to the respective S&A provided by an agent adopting the role of a specific CAPABILITY IMPLEMENTER on SEMANTIC HARDWARE LAYER. For this capability, the agent adopting the role of the CAPABILITY IMPLEMENTER then is responsible for:

- accessing the correct SDH providing the respective functionality as there may be more than one SDH connected to the agent (cf. the activity *access SDH functionalities* in Figure 3.2),
- executing the respective capability when this is required while acting in one of the other roles an agent adopts (cf. the activity *execute capability* in Figure 3.2, e.g., required by the CAPABILITY COORDINATOR in Figure 3.3),
- detecting that the execution of the associated capability has finished on the SDH (cf. the activity *stop capability execution* in Figure 3.2),
- stopping the capability's execution if necessary, i.e., in case the capability does not finish on its own like $c^{p}_{\text{MV-VEL}}$
- processing the callback of the respective SDH, i.e., being aware of the results the capability's execution produces, and
- making these results available for their further processing within the same or another role the agent adopts, i.e., the CAPABILITY COORDINATOR on AGENT LAYER or another CAPABILITY IMPLEMENTER on SEMANTIC HARDWARE LAYER in case of virtual capabilities.

Concerning capabilities agents can provide, we differentiate between virtual capabilities $\mathcal{C}^{v}$ and physical capabilities $\mathcal{C}^{p}$ on SEMANTIC HARDWARE LAYER. Both concepts refine the general concept of a capability. They encapsulate a parametrizable functionality that we can execute with hardware connected to the agent by addressing the respective CAPABILITY IMPLEMENTER. The difference between physical and virtual capabilities lies in the way they address the respective SDH. In comparison to physical capabilities, virtual capabilities are not directly associated with SDH. Instead, we require them to invoke associated other (physical) capabilities for their execution. Thus, virtual capabilities only have indirect access to SDH but can construct more complex behavior. Consequently, the set of parameters for a virtual capability needs to include additional information, e.g., the set of other capabilities it needs to combine for its execution which we note with the symbol ★. While there is no difference in using a physical or a virtual capability for higher-level layers, executing them on SEMANTIC HARDWARE LAYER makes a difference. In contrast to physical capabilities whose functionality a respective CAPABILITY IMPLEMENTER most often can call with a simple API access of the respective hardware driver implementation, executing a virtual capability requires more logic for correctly scheduling the access to different SDH and combining the results of them. We encapsulate this logic in the respective CAPABILITY IMPLEMENTER the agent adopts for making the virtual capability available for higher layers in the Multipotent System reference architecture.

Additionally, introducing virtual capabilities also has consequences for the self-awareness functionality of the SEMANTIC HARDWARE LAYER. For enabling the SELF-AWARENESS PROVIDER to interpret descriptions provided by SDH for physical capabilities correctly, we propose to use common dictionaries for defining their properties [Kosak et al., 2018; Wanninger et al.,

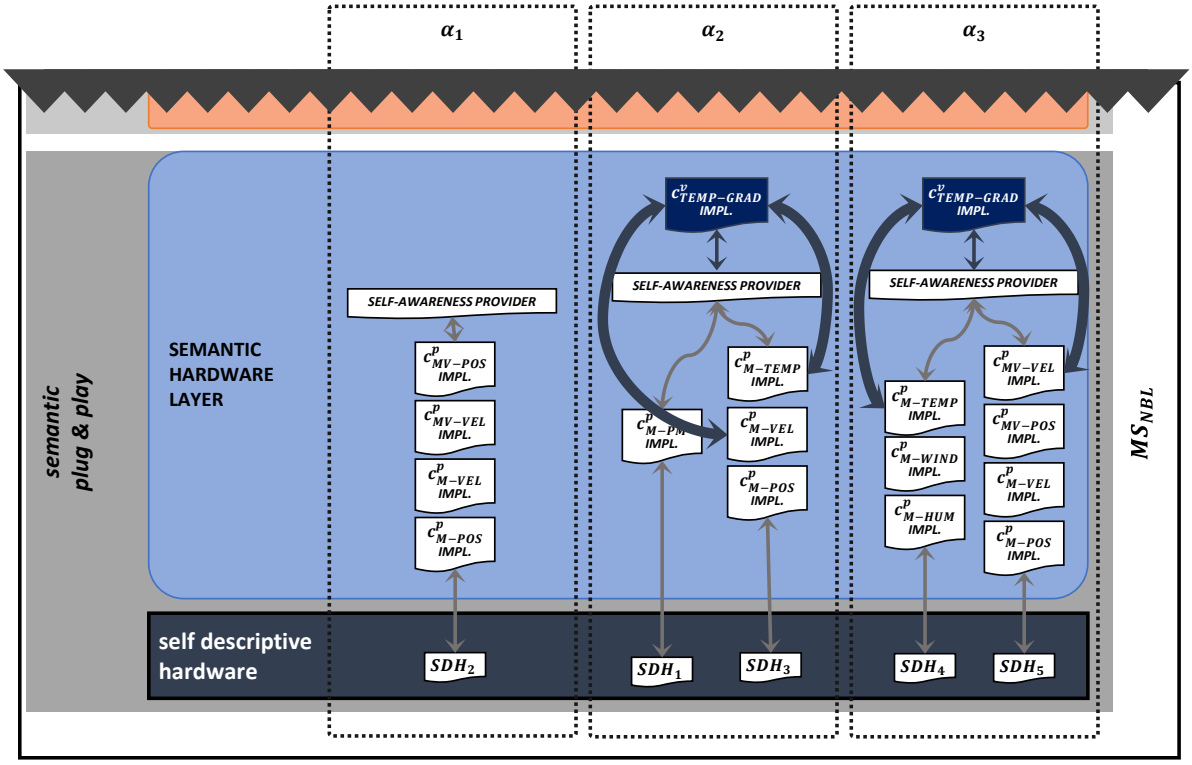


Figure 3.15: The virtual capability $c^v_{\text{TEMP-GRAD}}$ is available on SEMANTIC HARDWARE LAYER for agents α_2 and α_3 adopting the respective roles of $c^v_{\text{TEMP-GRAD}}$ IMPLEMENTER, because they provide the required physical capabilities $c^p_{\text{M-TEMP}}$ and $c^p_{\text{MV-VEL}}$ (cf. Table 3.9). Agent α_1 instead lacks an SDH providing the necessary functionality for the physical capability $c^p_{\text{M-TEMP}}$ and thus it also cannot adopt the role of a $c^v_{\text{TEMP-GRAD}}$ IMPLEMENTER. For the sake of clarity, we avoid depicting higher layers of the Multipotent System reference architecture in this figure.

2021]. Common dictionaries can help with that because we intend SDH to describe themselves in a stand-alone manner: For offering extensibility concerning new SDH and capabilities and maintaining modularity, we store information concerning the SDH directly on the devices in a distributed manner. Thus, no agent needs to know the specifics of capabilities if this is not relevant in its current physical configuration. This reduces the effort required for maintaining software deployed on agents, e.g., hardware drivers, and allows the AGENT LAYER to abstract from capability-specific details required to execute a capability on SEMANTIC HARDWARE LAYER. For virtual capabilities, we make an additional capability database available to the SELF-AWARENESS PROVIDER, providing the information relevant for their availability, i.e., which capabilities combine to virtual capabilities.

Running Example: Availability of Capabilities In our running example, we make use of physical capabilities and virtual capabilities. The functionality necessary for a physical capability's execution is directly provided by one SDH that provides that capability exclusively in its simplest form. In the configuration of agent α_1 in Figure 3.9, e.g., SDH_1 represents a sensor

for PM measurements only and thus also provides just one capability $c^p_{\text{M-PM}}$ for measuring the PM concentration. For these capabilities, we can easily make an agent adopting the respective role of a CAPABILITY IMPLEMENTER aware of which SDH it needs to access for actually executing the capability and how to access it. In Figure 3.9, e.g., agent α_2 in its role of the $c^p_{\text{M-TEMP}}$ IMPLEMENTER knows that for executing $c^p_{\text{M-TEMP}}$ it needs to address SDH_3 that encapsulates the respective S&A for measuring the current temperature.

Because SDH do not always represent simple S&A but can represent complex ones, they can also provide more than one capability to an agent they are assigned to. Thus, for the same SDH there might be multiple CAPABILITY IMPLEMENTERS on SEMANTIC HARDWARE LAYER. SDH_3 in Figure 3.9, e.g., represents an UAV offering complex onboard sensors like an Inertia Measuring Unit (IMU) in combination with an integrated sensor for temperature measurements (cf. Table 3.2). Being an UAV with an integrated temperature sensor, SDH_3 also encapsulates the functionality for executing $c^p_{\text{M-VEL}}$, $c^p_{\text{M-POS}}$, $c^p_{\text{MV-POS}}$, and $c^p_{\text{MV-VEL}}$. Thus, SDH_3 provides multiple capabilities to the agent it is assigned to, represented by different CAPABILITY IMPLEMENTERS on SEMANTIC HARDWARE LAYER of α_2. In all respective roles, i.e., that of a $c^p_{\text{M-VEL}}$ IMPLEMENTER, that of a $c^p_{\text{M-POS}}$ IMPLEMENTER, that of a $c^p_{\text{MV-POS}}$ IMPLEMENTER, and that of a $c^p_{\text{MV-VEL}}$ IMPLEMENTER, agent α_2 knows that it needs to address the same SDH. Agent α_3 instead knows that it needs to address different SDH for executing the capabilities it provides in it roles of a $c^p_{\text{M-TEMP}}$ IMPLEMENTER (SDH_4), that of a $c^p_{\text{M-POS}}$ IMPLEMENTER (SDH_5), that of a $c^p_{\text{MV-POS}}$ IMPLEMENTER (SDH_5), and that of a $c^p_{\text{MV-VEL}}$ IMPLEMENTER (SDH_5). For encoding this difference, each SDH describes them with properties provided to the agent.

Further, agents α_2 and α_3 also provide the virtual capability $c^v_{\text{TEMP-GRAD}} := \{c^p_{\text{MV-VEL}} \bigstar c^p_{\text{M-TEMP}}\}$ enabling them to move in a given direction until they detect a change in the dominant temperature gradient (like we require it in our running example in Section 3.2.7 for working on $\rho_{\text{NBL-S}'}$). The agents indicate this to CAPABILITY COORDINATOR on inter-agent level by adopting the roles of $c^v_{\text{TEMP-GRAD}}$ IMPLEMENTERs each (cf. Figure 3.15). The knowledge that they can adopt these roles is provided by the SELF-AWARENESS PROVIDER that has access to the application-specific capability database holding information on the virtual capabilities' requirements relevant for the use case (cf, Table 3.9). In our running example, the only entry here is $c^v_{\text{TEMP-GRAD}} := \{c^p_{\text{M-TEMP}} \bigstar c^p_{\text{MV-VEL}}\}$, indicating that for being able to provide $c^v_{\text{TEMP-GRAD}}$ by providing the respective role, an agent needs to also provide $c^p_{\text{M-TEMP}}$ and $c^p_{\text{MV-VEL}}$ by adapting the respective roles.

According to the properties provided by the set of SDH in combination with the application-specific database for virtual capabilities, each agent thus can determine which concrete capabilities it can provide. This way, in its role of a SELF-AWARENESS PROVIDER the agent can respond adequately, e.g., when requested by a PLAN BIDDER on AGENT LAYER (cf. Figure 3.8) during Ensemble Formation (cf. Figure 3.2)

3.2.8.2 Proposing and Executing Reconfigurations

To be able to provide the required information when a capability is not available to the agent, e.g., during a physical reconfiguration of the Multipotent System (cf. the activity *propose reconfiguration* in Figure 3.2), we propose to use the concept of *blueprints* [Wanninger et al., 2018]. Blueprints offer a declarative and semantically enriched interface describing capabilities and their properties. At run-time, we can *instantiate* and *execute* blueprints based on the

concrete available SDH or determine missing SDH in case a capability is not available. That way, we can abstract on AGENT LAYER whether addressing an CAPABILITY IMPLEMENTER addresses a single SDH, i.e., is a physical capability or addresses multiple SDH, i.e., is a virtual capability. Every capability, therefore, has a blueprint with a generic description of requirements that need to be fulfilled for making it available for an agent. Blueprints also offer necessary implicit pieces of information required to avoid instantiation that might provide the capability but restrict its execution, e.g., if an additional SDH connected to an agent harms the execution of capabilities provided by already connected SDH. The relevant relations between physical and virtual capabilities are stored in a common database with additional application-specific information, if necessary. During the process of Physical Reconfiguration, the agent thus needs to exploit information it has available through blueprints to propose only such new configurations that also enable it to not only have all required capabilities available but also ensure those capabilities are still executable in combination with all other capabilities and their respective SDH, e.g., taking into account the respective weights of SDH. Hence, the design of blueprints and how agents instantiate and execute them are responsible for achieving run-time heterogeneity of the Multipotent System being homogeneous at design-time.

Running Example: Blueprints and Their Instantiation We continue our running example describing the application of blueprints and how they can be instantiated for fulfilling the requirements in the described scenario. In our running example in Section 3.2.7, we require each agent adopting the roles of a PLAN WORKER IN $\rho_{\text{NBL-S}'}$ to be able to follow a temperature gradient when participating in the plan $\rho_{\text{NBL-S}'}$. Thus, for participating in the respective ensemble $\mathcal{E}^{\text{NBL-S}'}$, an agent needs to provide the respective capability $c^{v}_{\text{TEMP-GRAD}}$ that, according to the application-specific database for virtual capabilities, requires other capabilities to be available for its execution (cf. Table 3.9). Besides the capability for moving with a given velocity $c^{p}_{\text{MV-VEL}}$ also measuring the parameter of interest, i.e., measuring temperature with $c^{p}_{\text{M-TEMP}}$, is necessary. If the agent lacks providing one of the required capabilities (e.g., agent α_1 in Figure 3.10) it may need to provide a reconfiguration proposal (cf. the activity *propose reconfiguration* in Figure 3.2) in its role of a RECONFIGURATION PROPOSER on AGENT LAYER (cf. Figure 3.3). For delivering the correct information, therefore, it needs to be aware of how the user can adapt its physical configuration to fulfill the plan's requirements after that configuration. The agent can achieve this in its role of a SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER (cf. Figure 3.3). Regarding agent α_1 in Figure 3.15, we can see that in addition to SDH_2 requires another SDH. While SDH_1 provides the capability $c^{p}_{\text{MV-VEL}}$ to α_1 so that it can adopt the role of a $c^{p}_{\text{MV-VEL}}$ IMPLEMENTER on SEMANTIC HARDWARE LAYER, it cannot provide the capability $c^{p}_{\text{M-TEMP}}$ in its current configuration. Thus agent α_1 cannot provide the virtual capability $c^{v}_{\text{TEMP-GRAD}}$. It, therefore, requires an SDH encapsulating a temperature sensor so that it also can adopt the role of a $c^{p}_{\text{M-TEMP}}$ IMPLEMENTER. When selecting such an additional SDH, the α_1 needs to take into account the properties of the set $\mathcal{SDH}_{\alpha_1}$ it currently has connected as well as those that it might have connected after a successful reconfiguration. In combination, $\mathcal{SDH}_{\alpha_1}$ can have a too high overall weight, which might exceed the maximum possible weight the UAV can carry while flying (we elaborated on the consequences for generating reconfiguration proposals in the course of our running example in Section 3.2.7).

In Figure 3.16, we illustrate such a situation. Agent α_1 tries to instantiate the blueprint

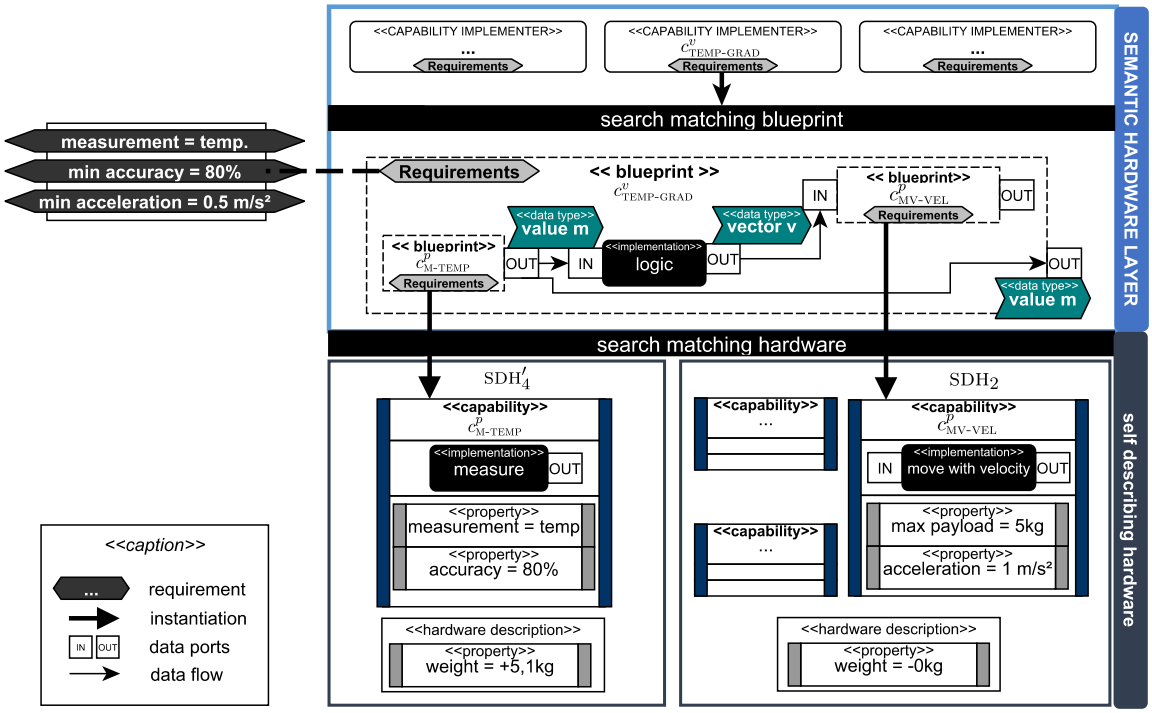


Figure 3.16: Situation during a reconfiguration of $\mathcal{MS}_{\text{NBL}}$ for $\rho_{\text{NBL-S}}$ focusing on agent α_1 in a configuration as depicted in Figure 3.15. Agent α_1 cannot provide the required physical capability $c^p_{\text{M-TEMP}}$ and thus not the virtual capability $c^v_{\text{TEMP-GRAD}}$. Therefore, it searches for possible instantiations of the blueprint for $c^v_{\text{TEMP-GRAD}}$. A configuration including SDH'_4 additional to SDH_2 would not suffice the requirements, because the total weight of α_1 would exceed the maximum weight the agent is still able to move, i.e., execute $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-VEL}}$. Thus, in its role of a SELF-AWARENESS PROVIDER (cf. Figure 3.3), agent α_1 requires to find an alternative configuration enabling it to provide $c^p_{\text{M-TEMP}}$ without restricting the availability of capabilities already provided by SDH_2.

for the capability $c^v_{\text{TEMP-GRAD}}$ with SDH_2 it already is configured with and an additional $\text{SDH}_{4'}$. In contrast to SDH_4, we modify the weight of the SDH only slightly from $5kg$ to $5.1kg$ for this example, having an immense impact on the available capabilities of α_1. While both capabilities seem to be available to the agent in the given configuration, and thus also the capability $c^v_{\text{TEMP-GRAD}}$ should be available to the α_1, the agent cannot execute all of these capabilities. Caused by the too-high total weight, the agent can no longer execute its capability of moving $c^p_{\text{MV-VEL}}$. While this does not influence the capability for measuring $c^p_{\text{M-TEMP}}$, the capability for executing the temperature-based gradient flight relying on $c^p_{\text{MV-VEL}}$ also no longer can be executed. Thus, e.g., when generating a reconfiguration proposal, agent α_1 instead suggests being configured with SDH_4 instead of $\text{SDH}_{4'}$, allowing it to provide $c^p_{\text{M-TEMP}}$ without restricting its other capabilities so that it can also provide the virtual capability $c^v_{\text{TEMP-GRAD}}$.

3.3 A Reference Implementation Prototype Using the Jadex Framework

For evaluating and demonstrating the feasibility and functionality of the concepts we introduce and describe in Section 3.2, we implemented a prototypical Multipotent System with the Jadex Active Components Framework (Jadex) developed by Braubach and Pokahr [2012] for realizing distributed Multi-Agent Systems. Thus, the following section handles the details of this implementation, describing the software technical solution we apply and is intended for the reader interested in constructing its own reference implementation of a Multipotent System. We use the prototypical implementation we present here to evaluate the concepts of Multipotent System in the following Section 3.4.

Introducing the concept of Jadex Platforms, Jadex allows for the distribution of applications deployed on different devices. A developer can host each Jadex Platform as required by its application on the same or a different device, offering the possibility of hosting a Java Virtual Machine. Each platform runs in its separate Java process that can communicate with other Jadex Platforms running on the same device or other devices within the same network infrastructure. In addition to offering this communication middleware for agents, Jadex provides the possibility to dynamically load and unload so-called *Active Components* from Jadex Platforms. We exploit this feature to realize the core concept of our approach. We represent each role an agent can adopt within our Multipotent System reference architecture as such an *Active Component*. The relevant sub-type of an *Active Component* we use in our approach is called Micro-Agent. To avoid possible obfuscation in this section, we refer to these agents as Micro-Agents and agents from our Multipotent System reference architecture as Multipotent-Agents for the rest of this section. Using Micro-Agents allows us for realizing all roles specifically relevant for all different functions and situations the Multipotent System and its included Multipotent-Agents are situated in, i.e.,

- creating plans and allocating plans to Multipotent-Agents with the roles adopted by the user and by Multipotent-Agents on PLAN LAYER,
- forming ensembles that can handle those plans by the interplay of the roles adopted by Multipotent-Agents on ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER,
- cooperatively working on plans by coordinating ensembles in the roles Multipotent-Agents adopt on ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER,
- suggesting adaptations of the Multipotent System's configuration with the roles adopted by Multipotent-Agents on ENSEMBLE LAYER, AGENT LAYER, and SEMANTIC HARDWARE LAYER, and
- adapting the set of capabilities $\mathcal{C}_\alpha$ an agent provides with the roles adopted on SEMANTIC HARDWARE LAYER in combination with the concept of Self-Descriptive Hardware (SDH) and the interactions with the user.

To allow for the desired modularity and adaptability of our approach, we realize all three main components as separate Jadex Platforms:

- We represent each Multipotent-Agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ as a Jadex Platform enabling them to communicate with each other when operating within the same network.
- We represent each SDH $\in \mathcal{SDH}_{\mathcal{MS}}$ as a Jadex Platform enabling Multipotent-Agents and SDH to communicate with each other when running on devices within in the same network.

- We integrate the user's device in the Multipotent System as a Jadex Platform enabling it to communicate and interact with Multipotent-Agents and SDH running on devices within the same network.

In the rest of this section, we describe the elements from Jadex relevant for our prototypical implementation and investigate the concepts we use to realize the respective platforms for Multipotent-Agents, SDH, and the user's device.

3.3.1 The Jadex Active Components Framework

Jadex is a Java framework designed to facilitate the construction and implementation of distributed systems based on the concept of *Active Components* [Braubach and Pokahr, 2012]. For this purpose, Jadex provides various Java classes, Java interfaces, and Java annotations. Developers of distributed systems can use those concepts to transform regular Java classes into Active Components providing services to other Active Components for enabling their communication. According to Braubach and Pokahr [2012], in Jadex the components of a distributed system implemented as *Active Components* can be divided into *components*, *services* (*provided services* and *required services*), and *sub-components*. Jadex supports different implementations of such components, e.g., the modeling and implementation of systems using the Belief, Desire, Intention (BDI) paradigm or the modeling and implementation of systems using the Business Process Modeling Notation (BPMN) among others. In our explanations here, we focus on the concept of so-called *JadexMicro-Agents* which we use to realize the concepts of our Multipotent System reference architecture from ṛefsec:multipotentsystems-architecture.

3.3.1.1 Description of a Jadex Active Component and Jadex Platforms

Micro-Agents are *Active Components*. They can be launched on so-called Jadex Platforms by the Jadex framework. Jadex Platforms provide the ability to launch multiple Micro-Agents on them. Developers can make use of this feature to modularize and parameterize the system. Micro-Agents provide an execution platform for externally provided services (the Micro-Agent's individually *Provided Services*). For acting as intended by the developer, Micro-Agents can require to access the provided services of other Micro-Agents (the *Required Services* of the Micro-Agent). Developers can use this feature to design and implement asynchronous and distributed systems using different types of Micro-Agents that can provide different types of services each. To achieve a higher degree of modularity during system design, developers can subdivide components like Micro-Agents into sub-components. Sub-components can encapsulate individual functionality and use the Micro-Agent's communication interface to also provide services to other Micro-Agents (or their sub-components) and require services provided by other Micro-Agents (or their sub-components). When started on a Jadex Platform, each Micro-Agent undergoes a life-cycle the system's developer can individually define for each type of Micro-Agent during system design. The life-cycle consists of the three steps *initialization*, *lifetime*, and *shutdown*.

- **Initialization State**: While a Micro-Agent already can request the services of other Micro-Agents already being in their lifetime state, its provided services are not yet accessible for other Micro-Agents during the Micro-Agent's *initialization*. A developer thus can use the initialization state to setup each Micro-Agent's internal state correctly and initialize the Micro-Agent's communication with other Micro-Agents. After finishing all

procedures defined in its initialization block, the Micro-Agent automatically switches its state to *lifetime*.

- **Lifetime State**: While being in its lifetime state, a Micro-Agent provides its services to other Micro-Agents requiring them. Further, the Micro-Agent can request other Micro-Agents to execute their provided services if required by the application. Thus, the lifetime state represents the actual active time of the *Active Component*.
- **Shutdown State**: While being in its shutdown state, the Micro-Agent does no longer provide its services for other Micro-Agents. Similar to the initialization state, the Micro-Agent can still access provided services of other Micro-Agents. Developers thus can use the shutdown state to dissolve previously established communication structures letting the Micro-Agent actively inform other Micro-Agents it communicated with while being in its lifetime state.

Using the Micro-Agent life-cycle as intended can help developers using the Jadex framework to avoid undefined behavior of the system they develop that otherwise could emerge when letting Micro-Agents interact in non-productive states.

3.3.1.2 Interaction of Active Components Through Provided Services

Services are interfaces implemented by Micro-Agents directly or by one of the Micro-Agent's sub-components. If a Micro-Agent implements a specific service, this service is provided to other Micro-Agents in the system that they can make use of it. Developers can use this feature of Jadex to create a rough abstraction of the overall system: If different Micro-Agents provide the same service, i.e., implement the same interface, each of these Micro-Agents can still use a different implementation of the respective service. For example, two different Micro-Agents might provide the service s of moving the same robot to a given position. While the interface accessible to other Micro-Agents provided by the service s stays the same, the underlying implementation might differ. While a Micro-Agent requiring and calling the service s can rely on the robot moving to the given location when calling the service s provided by one of the Micro-Agents, i.e., the robot finally reaches the desired position, it has no information on further details on the service's concrete implementation. How the robot reaches the position is hidden by the respective Micro-Agent's implementation of the service s that might differ from one Micro-Agent to another, e.g., in the routing algorithm used for navigating to the position. Thus, defining interactions between Micro-Agents with provided and required services enables the developer of a distributed system to ensure modularity during system design.

3.3.1.3 Finding Provided Services with Service Discovery

Searching and addressing provided services is managed by the Jadex framework internally. Thus, a developer can specify and implement required and provided services of Micro-Agents without having deep knowledge of the actual communication between Micro-Agents. During designing their system with the Jadex framework, developers can focus on their actual application. This often requires deciding for the correct service to be called by a Micro-Agent. Developers, therefore, need to define and restrict the availability of services provided to other Micro-Agents. To channel the service search correctly, the Jadex framework provides appropriate abstractions to developers. A developer can specify different relevant aspects defining how to search for required services and how to provide services to other Micro-Agents in the system for every Micro-Agent individually.

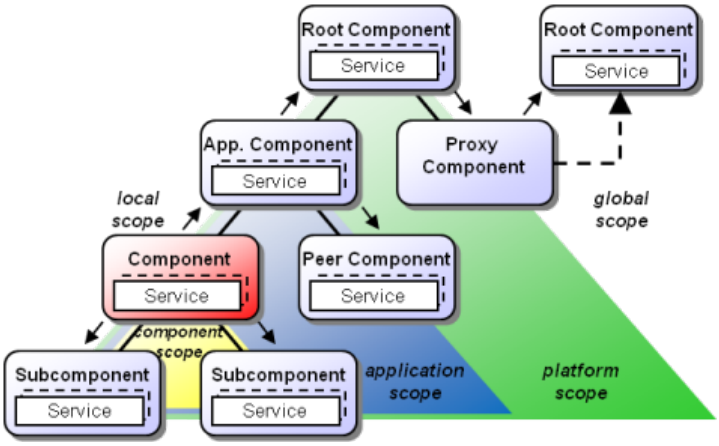


Figure 3.17: The differnet scopes possible during the search for provided services performed by *Active Components* like Micro-Agents in the Jadex framework. Figure from [Alexander Pokahr and Jander, 2018].

- **Scope of the service search**: Before starting the search for providers of its required services, a Micro-Agent can specify the range of this search. Possible ranges include taking into account the Micro-Agents sub-components only (*component scope*), taking into account all other Micro-Agents running on the same Jadex Platform (*platform scope*), or even taking into account all Micro-Agents running on the same or other Jadex Platforms that can be reached within the network (*global scope*). We depict the different scopes in Figure 3.17, taken from Alexander Pokahr and Jander [2018]. Because other Jadex Platforms can be hosted on the same device, other devices within the same local network, or even within the whole internet, the Jadex framework provides powerful possibilities for most applications.
- **Dynamic or Static Search**: When a Micro-Agent searches for a required service more than once during its lifetime, a *dynamic search* triggers a new search within the defined scope every time before the service(s) invocation. Before doing so, a static search instead first tries to invoke services found during previous searches performed by the Micro-Agent, managed in cache storage. This enables developers to detect and use new service providers in the system if required by the application.
- **Calling All or Only Specific Services**: When a Micro-Agent is aware of the possible providers of the services it requires, the Micro-Agent can specify whether to call all of these services, only call specific services, or filter services according to other criteria, e.g., the type, name, or state of the service's provider. This allows the developer to access precisely these services the application requires in a specific situation.
- **Synchronous and Asynchronous Communication**: When calling a required service, the developer can define whether the communication between the service provider and service requester should be handled synchronously or asynchronously. With a synchronous service call, the developer can require the caller of a service to wait for the callback of the service's provider before allowing the caller to proceed with its program execution. With an asynchronous service call, the developer can allow the caller to proceed with its execution and receive the callback independently.

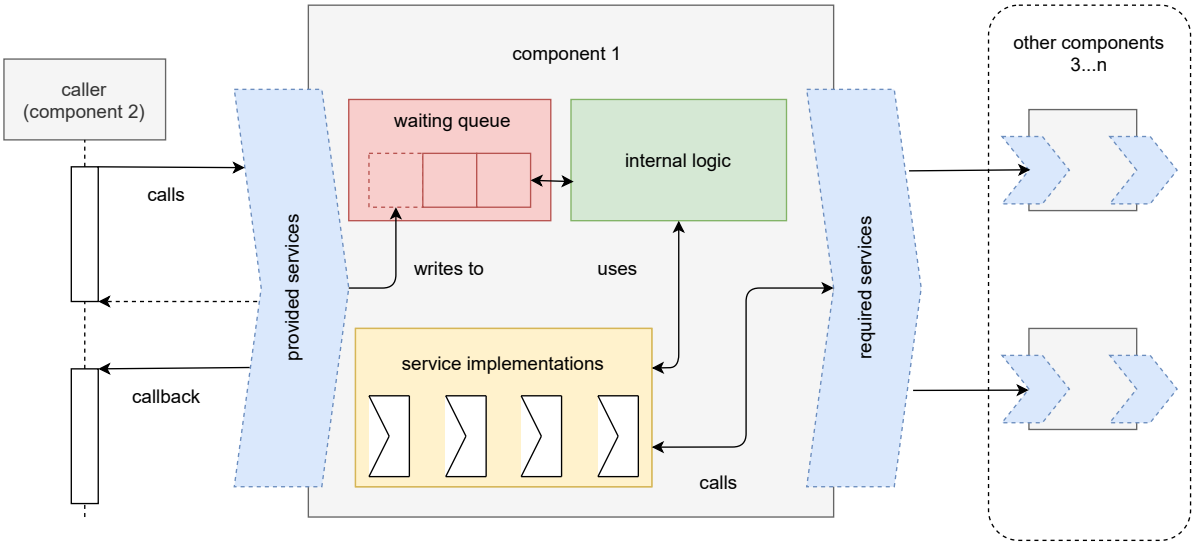


Figure 3.18: The structure of an active component *component 1* in Jadex focusing on the relevant parts during an externally triggered call from another component *component 2*, including the interactions with other active components *component 3, ..., n*. First, one of the provided services *service 1* of *component 1* is accessed by the call of *component 2*. Second, the call from *component 2* is inserted into a waiting queue where other calls from the same component *component 2* or other components *component 3, ..., n* wait to be further processed by *component 1*. Third, for the further processing of calls waiting in the waiting queue, *component 1* calls its internal implementation for the respective service *service 1*. Fourth, working through this implementation can require *component 1* to call services provided by other external components, i.e., *required services* from the scope of *component 1*. Fifth, after completely processing a call from the waiting queue including all possible external calls of required services provided by *component* $3, ..., n$, the result is passed back to *component 2* asynchronously. Figure adapted from Eing [2020].

3.3.1.4 Accessing Provided Services with Service Invocation

Addressing a service with Jadex now proceeds as follows (we illustrate the interrelationships of interactions performed between Active Components in Figure 3.18): The specific service is searched for within the defined scope using a search mechanism managed by the Jadex framework. During the process of developing a Micro-Agent, the respective services are delivered to the developer as an implemented interface. At run-time, the communication middleware provided by Jadex then delivers the calls of methods provided by this interface to the selected Micro-Agent(s) implementing the respective service. Micro-Agents receiving service invocations then insert the calls into a waiting queue for preparing their further processing. Calls waiting in the waiting queue then are processed by the service's implementation, defined by the developer for the respective Micro-Agent. In this implementation, the developer of the Jadex application may include other services required for execution. This can create a chain of service calls using multiple Micro-Agents and their provided services. Once a Micro-Agent has finished processing one invocation of a call of one of its provided services, it returns the value to the Micro-Agent initially calling the provided service as defined by the service's interface.

To prevent waiting for responses from other Micro-Agents before being able to continue its execution further (e.g., responding to a call directed to one of its provided services), Micro-Agents in Jadex operate according to the *Actor Model* [Hewitt et al., 1973]. By introducing the possibility of asynchronous communication, using the Actor Model can avoid inefficient programs with long waiting times and even prevent deadlocks caused by bidirectional communication.

3.3.1.5 Conclusion

Jadex provides understandable abstractions, intuitive operation and reduces complexity in handling distributed systems. Thereby, the Jadex framework allows the developer to focus on implementing the desired behavior of its application. In previous work, the use of *Active Components* already has proven to be a valuable tool for the development of both physical distributed systems and virtual MAS/MRS [Seebach et al., 2007]. All these aspects qualify the Jadex framework for the prototypical implementation of the Multipotent System reference architecture we require for validating and evaluating our concepts introduced in Section 3.2.

3.3.2 Mapping Concepts From Multipotent Systems to Concepts in Jadex

To realize our Multipotent System reference architecture with the Jadex framework, we need to map concepts appropriately. While making use of the Jadex-concepts for a prototypical implementation, we keep in mind different aspects of relevance:

- The implementation must run on low-performance computers to deploy it to flying ensembles that only have limited computing power onboard.
- The implementation must support the flexible exchange of hardware modules in an Multipotent-Agent's SDH configuration.
- The implementation must be extensible concerning new applications requiring new capabilities in new use-cases.
- The implementation should also be executable in a simulated environment, enabling prototyping of new applications and capabilities.

In the following, we describe how we can achieve this for Multipotent-Agents, SDH-prototypes, and the user's device, creating an extensible and highly flexible prototypical implementation of a Multipotent System.

3.3.2.1 Initializing and Deploying Jadex Platforms

For realizing the required distribution of our prototypical implementation of the Multipotent System reference architecture, we designed a common Jadex Platform start-up routine. We use this routine for starting all types of Jadex Platforms within our prototypical implementation, i.e., we use it for *Multipotent-Agents*, for *different* SDH *-prototypes*, and for the *user's device*. Doing so ensures that the communication between parts of the application running on a different host, i.e., different hardware, is possible without most technical hurdles (cf. Figure 3.21 and Figure 3.22). For achieving this, we create an abstract Jadex Platform starter (cf. the Java Class *AbstractPlatformStarter* in Listing 3.1).

We use such because every Jadex Platform needs to be parametrized correctly enabling it to detect other Jadex Platforms for itself and for being visible to other Jadex Platforms running on other hosts within the same network. By executing the main method of the *AbstractPlatformStarter* (cf. Listing 3.2, Line 12) we describe in detail in Listing 3.2, we already set the

```
1  // Class used to start a new platform with appropriate configuration.
2  public abstract class AbstractPlatformStarter {
3    // the Micro-Agents to be started on the Jadex Platform
4    private List<Tuple2<String, CreationInfo>> asts;
5    // ID for auto-connecting Micro-Agents and self-descriptive hardware
6    private String autoconnectID;
7    // local address of the start-up helper Micro-Agent
8    public static String StarterMicroAgent = StarterMicroAgent.class.getName() +
   ".class";
9
10   // Main function for starting the Platform
11   protected void startPlatform(String[] args) {
12     // cf. the details presented in the additional Listing
13     ...
14   }
15
16   // starts the necessary Micro-Agents for this Jadex Platform
17   protected void startAgents(IComponentManagementService cms) {
18     // initialize the list of Micro-Agents to start on the Jadex
   Platform
19     this.asts = new ArrayList<Tuple2<String, CreationInfo>>();
20     // load the Jadex Platform's specific Micro-Agents into asts
21     loadAgentsToStart();
22     // create start-up info for the start-up helper Micro-Agent
23     CreationInfo agents = new CreationInfo(SUtil.createHashMap(new String[] { "agents" },
   new Object[] { asts }));
24     // start the helper Micro-Agent that starts all other Micro-Agents
25     cms.createComponent(StarterMicroAgent, agents);
26   }
27
28   // abstract method to load the list of specific Micro-Agents
29   protected abstract void loadAgentsToStart();
```

Listing 3.1: *AbstractPlatformStarter*

Figure 3.19: Class *AbstractPlatformStarter* each other Jadex Platform in our application extends to ensure their compatibility and enabling their undisturbed communication.

most relevant parameters within a Jadex Platform's configuration. In Line 21 to Line 35 of Listing 3.2, we first create a new default platform configuration and subsequently modify its default parameters. By setting the *trustedLan* parameter to false (Line 23 in Listing 3.2), we first generally exclude all other platforms running in the same network to be able to detect and call provided services of the platform we currently start. That way, we can have multiple instances of our prototypical implementation running within the same network without interfering with each other. This is important, especially for testing and development purposes that often require to be run in parallel. To enable cross-platform communication within the same network, we then set a *networkName* and a *networkPassword* (Line 25 and Line 27 of Listing 3.2), letting all Micro-Agents running on platforms that share this information communicate with each other by calling their respective provided services. In Line 29 of Listing 3.2 we set the default timeout for such asynchronous service calls initiated by Micro-Agents running on the

Jadex Platform. Thereby, we specify how long we want to wait for a response to the service call until we assume that the corresponding Micro-Agent will no longer answer the request. By setting multiple awareness parameters (i.e., *awareness*, *awaDelay*, and *awaMechanism* in Line 31 to Line 35 in Listing 3.2), we configure the Jadex Awareness Mechanism for detecting other platforms in the network automatically and in a as-robust-as-possible way (i.e., by enabling all relevant Jadex Awareness mechanisms *broadcast*, *multicast*, *local*, and *message*). In Line 38 in Listing 3.2, we then start a new Jadex Platform using the created configuration and wait for it to finish its start-up sequence by calling the Jadex-specific *IFuture<>.get()* method in Line 40 in Listing 3.2 to synchronize the originally asynchronous call. After successfully launching the Jadex Platform, we then begin with starting the necessary Micro-Agents we want to run on the Jadex Platform by first retrieving the Jadex Platform's component management service in Line 42 in Listing 3.2 and subsequently initializing the actual starting of Micro-Agents in Line 44 in Listing 3.2. We let every concrete Jadex Platform, i.e., Jadex Platforms for *Multipotent-Agents*, SDH-*prototypes*, and the *user's device*, extend this abstract starter, ensuring their compatibility and enabling their undisturbed communication. For each Jadex Platform, we load the set of Micro-Agents *asts* we want to run on that platform (cf. Listing 3.1, Line 4) with the abstract method *loadAgentsToStart* (cf. Listing 3.1, Line 21) implemented concretely by the specific Jadex Platform (cf. Listing 3.1, Line 29). Thereby, we can individualize each Jadex Platform by starting the specific Micro-Agents implementing the required roles for that platform (cf. Listing 3.1, Line 25) like we define them in our Multipotent System reference architecture.

For setting-up Jadex Platforms having one or multiple SDH connected at start-up more easily, we introduced an identifier for connecting SDH automatically on start-up (cf. autoconnectID Listing 3.1, Line 7) which we can add to the program's arguments when initializing the Jadex Platform. We can use this, e.g., for a Multipotent-Agent α having multiple SDH configured in its $\mathcal{SDH}_\alpha$, which we want to set up already having the necessary knowledge about this state. All Jadex Platforms we start with the same value for this identifier connect automatically with a dedicated mechanism. This mechanism is proper when hosting our prototypical implementation on different devices in a distributed manner (cf. Figure 3.22). When doing so, we do not necessarily have direct access to all these devices for manually performing the Multipotent-Agent's configuration. Further, when scaling the application, manually configuring the Multipotent System (i.e., connecting all SDH $\in \mathcal{SDH}_{\mathcal{MS}}$ to all $\alpha \in \mathcal{A}_{\mathcal{MS}}$ as required) is very time-intense and would heavily reduce productivity.

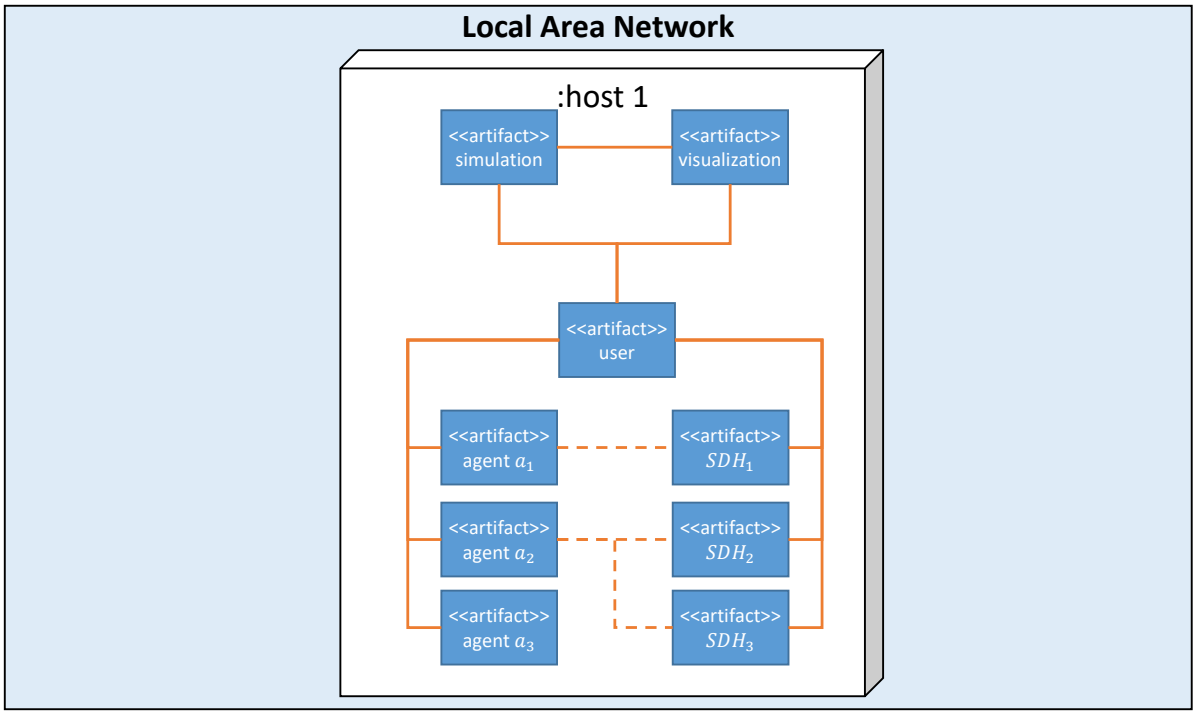
We further use the abstract starter to host additional Jadex Platforms we only require when running our prototypical implementation in a virtual environment (i.e., when not using real S&A hardware). Therefore, we implemented Jadex Platforms providing the necessary functionality for the simulation and visualization of a Multipotent System. We designed such for embedding an extended version of the Robotics Application Programming Interface (RAPI) [Angerer et al., 2013] simulation environment and another for an adapted version of the ROS framework [Quigley et al., 2009]. The possibility for the auto-configuration of the Multipotent-Agents using the *autoconnectID* (cf. Listing 3.1, Line 7) is also very useful in that virtual environments. We can run multiple Jadex Platforms on the same host device while we are still able to differentiate between different Multipotent-Agents and SDH instances.

We depict multiple possible deployments of Jadex Platforms encapsulated in individual software «artifacts» that we suppose to be useful for simulating Multipotent System in Figure 3.21 and for deploying Multipotent System to real hardware in Figure 3.22. We assume an exemplary Multipotent System consisting of α_1, α_2, and α_3 configured with SDH_1, SDH_2, and

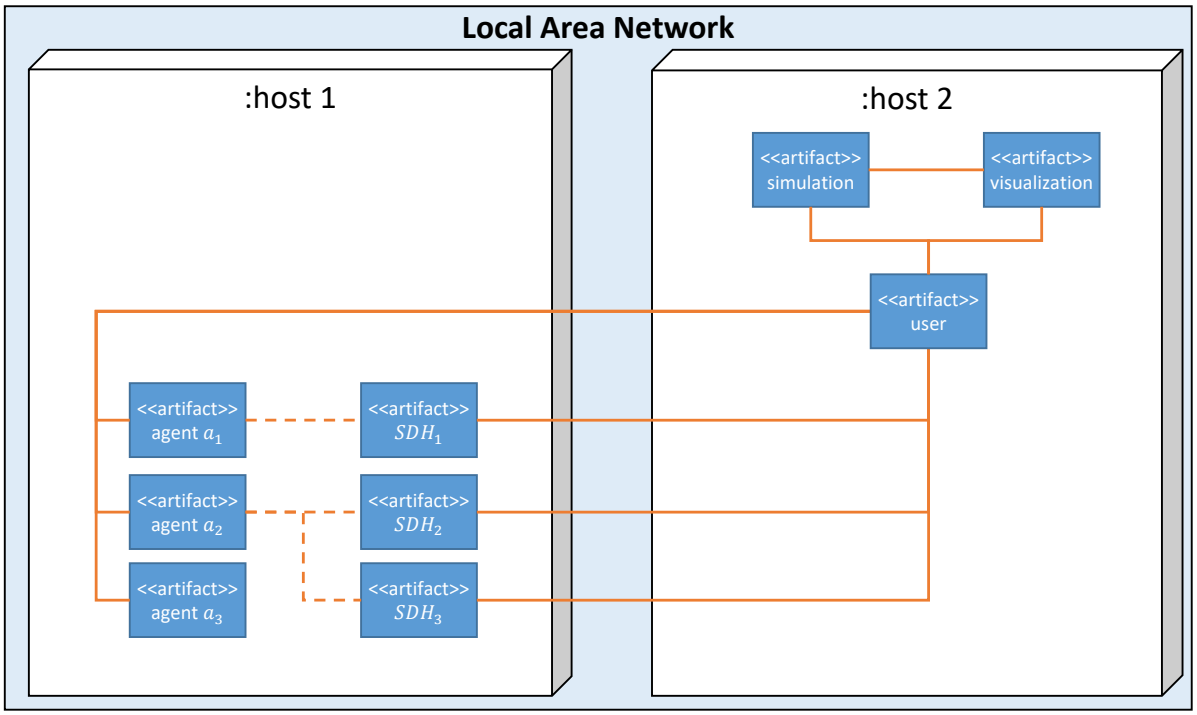
```
1  // Main function for starting the Platform
2  protected void startPlatform(String[] args) {
3    // load the program arguments as a list for further processing
4    List<String> programArgs = Arrays.asList(args);
5
6    try {
7      // load the auto-connect ID to connect Micro-Agents and SDH
8      if (programArgs.contains("-id")) {
9        this.autoconnectID = programArgs.get(programArgs.indexOf("-id") + 1);
10       System.out.println("Using auto-connect with id " + this.autoconnectID);
11     }
12     // handle other program arguments if available
13     else { ... }
14     // load parameters from a file initializing Environment
15     GeneralConceptParameters.class.getConstructor().newInstance();
16   } catch (Exception e) {
17     e.printStackTrace();
18   }
19
20   // load Jadex internal default configuration
21   PlatformConfiguration platConf = AbstractplatConfuration.getDefault();
22   // trust other Jadex Platforms in the same local area network
23   platConf.setTrustedLan(false);
24   // define a Jadex sub-network to allow for correct service scoping
25   platConf.setNetworkName(Environment.parameters().APPLICATION_NAME);
26   // define a password all Jadex Platforms from the application share
27   platConf.setNetworkPass(Environment.parameters().APPLICATION_NAME);
28   // define a timeout for asynchronous communication
29   platConf.setDefaultTimeout(Environment.parameters().DEFAULT_TIMEOUT);
30   // enable detection of other Jadex Platforms in the network
31   platConf.setAwareness(Environment.parameters().ENABLE_PLATFORM_AWARENESS);
32   // define how long to search for other Jadex Platforms
33   platConf.setAwaDelay(Environment.parameters().PLATFORM_AWARENESS_DELAY);
34   // define how to search for other Jadex Platforms in the network
35   platConf.setAwaMechanisms(RootComponentConfiguration.AWAMECHANISM.broadcast,
     RootComponentConfiguration.AWAMECHANISM.multicast,
     RootComponentConfiguration.AWAMECHANISM.local,
     RootComponentConfiguration.AWAMECHANISM.message);
36
37   // create a Jadex Platform with the modified start configuration
38   IFuture<IExternalAccess> platformfut = Starter.createPlatform(platConf);
39   // wait for the Jadex Platform to be created (synchronous call)
40   IExternalAccess platform = platformfut.get();
41   // load the component management service to start Micro-Agents
42   IComponentManagementService cms = SServiceProvider.getService(platform,
     IComponentManagementService.class, RequiredServiceInfo.SCOPE_PLATFORM).get();
43   // start Jadex Platform's relevant Micro-Agents
44   startAgents(cms);
45 }
```

Listing 3.2: *AbstractPlatformStarter.startPlatform*

Figure 3.20: Main function from the class *AbstractPlatformStarter* we use to configure important Jadex-specific settings.

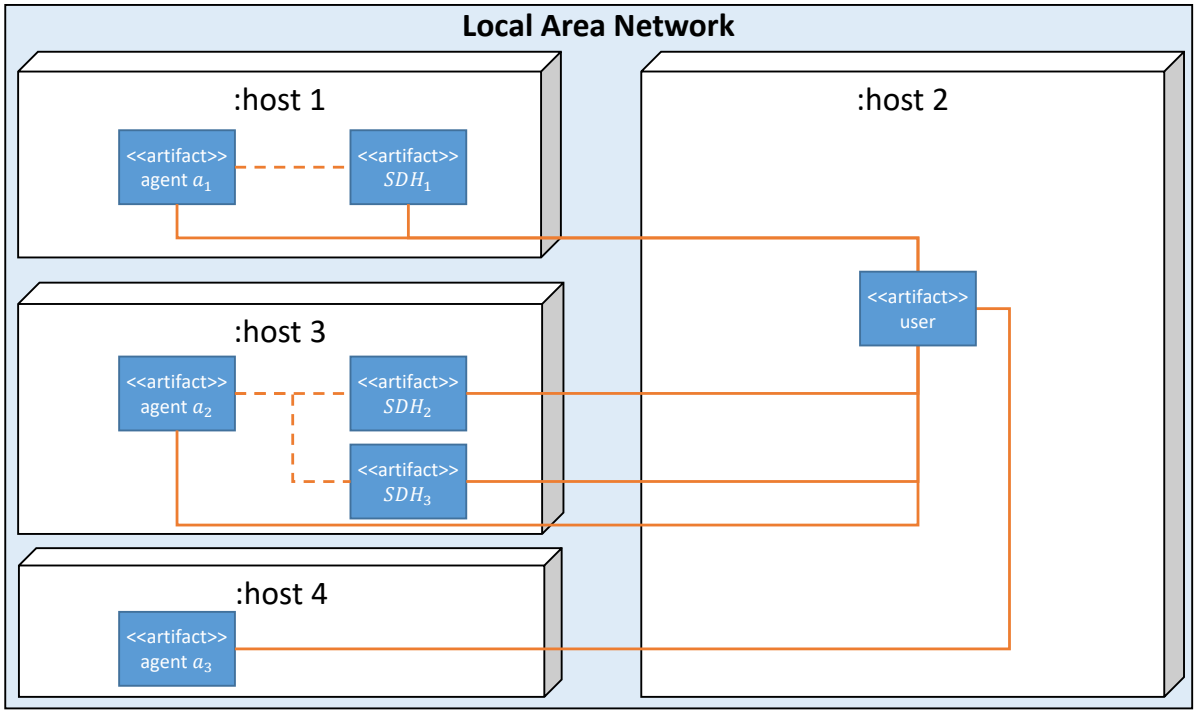


(a) possible deployment for simulation purposes on only one host

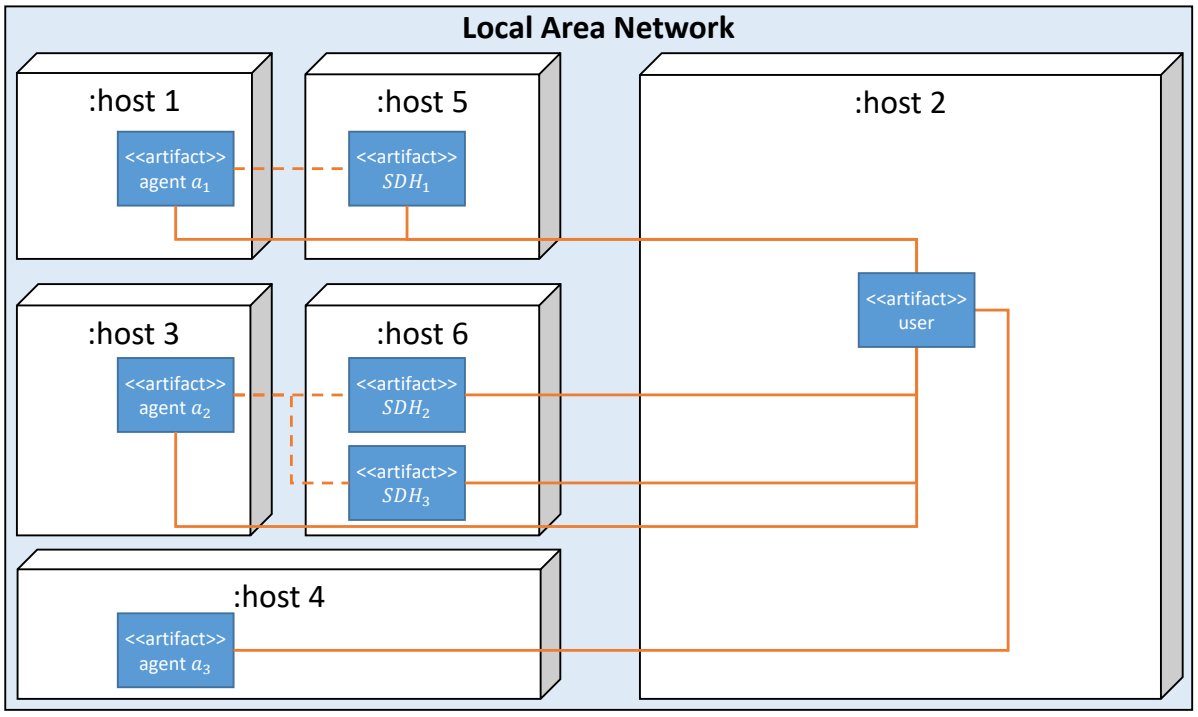


(b) possible deployment for simulation purposes on multiple hosts

Figure 3.21: Possible deployments of Jadex Platforms, i.e., independent software «artifacts», on one (Figure 3.21a) or multiple (Figure 3.21b) hosts when simulating the Multipotent System consisting of Multipotent-Agents α_1, α_2, and α_3 configured with SDH_1, SDH_2, and SDH_3 so that $\text{SDHs}_{a_1} := \{ \text{SDH}_1 \}$, $\text{SDHs}_{a_2} := \{ \text{SDH}_2, \text{SDH}_3 \}$, and $\text{SDHs}_{a_3} := \{ \}$, using a simulation and a visualization environment. Connections between «artifacts» indicate communication between Micro-Agents running on the respective Jadex Platforms. Dashed connections visualize Multipotent-Agent configurations that influence a Multipotent-Agents capabilities.



(a) possible deployment for real-world usage with one host per Multipotent-Agents



(b) possible deployment for real-world usage with sperate hosts for sdh

Figure 3.22: Possible deployments of Jadex Platforms, i.e., independent software «artifacts», when deploying the Multipotent System ti real hardware, consisting of Multipotent-Agents α_1, α_2, and α_3 configured with SDH_1, SDH_2, and SDH_3 so that $\text{SDHs}_{a_1} := \{ \text{SDH}_1 \}$, $\text{SDHs}_{a_2} := \{ \text{SDH}_2, \text{SDH}_3 \}$, and $\text{SDHs}_{a_3} := \{ \}$ to real hardware. Connections between «artifacts» indicate communication between Micro-Agents running on the respective Jadex Platforms. Dashed connections visualize Multipotent-Agent configurations that influence a Multipotent-Agents capabilities.

SDH_3 so that $\mathcal{SDH}_{a_1} := \{ \text{SDH}_1 \}$, $\mathcal{SDH}_{a_2} := \{ \text{SDH}_2, \text{SDH}_3 \}$, and $\mathcal{SDH}_{a_3} := \{ \}$. Deploying a whole Multipotent System including all Jadex Platforms on a single host *host 1* as we depict in Figure 3.21a helps during the creation of new prototypes or for integrating new concepts during the development phase of a specific Multipotent System. In this use-case, we also need to host independent software «artifacts» for *simulation* and *visualization*. If necessary, e.g., for increasing computational power in large scale applications, we can also deploy different «artifacts» (i.e., Jadex Platforms) on multiple hosts using the possibility of distributing Jadex applications (cf. Figure 3.21b). With the design we propose for our prototypical implementation of the Multipotent System reference architecture based on Jadex concepts, we can distribute our application to even more than only two hosts as depicted in Figure 3.21b. We can use one host per Jadex Platform, also enabling large-scale Multipotent System if required. Figure 3.22b comes close to such deployment but is dedicated to deploying the Multipotent System to real hardware. We, therefore, do not require hosting a software «artifact» for a simulation or a visualization. Instead, *host 2* in Figure 3.22 only hosts the run-time environment for the user's device, i.e., the *«artifact» user* encapsulating the respective Jadex Platform. We can let other Jadex Platforms run on individual hosts each or group them on multiple hosts as required and according to the processing power of the available host. In Figure 3.22a, we deploy Jadex Platforms for different Multipotent-Agent on individual hosts, each side by side with the respective Jadex Platforms for an SDH from their $\mathcal{SDH}_\alpha$. If possible, according to the respective hardware and if useful for the respective application, we can also deploy these Jadex Platforms on a different host. In Figure 3.22b, e.g., we host *«artifact» Multipotent-Agent* α_1 on a different host (*host 1*) than the Jadex Platform encapsulated in «artifact» SDH_1 (*host 5*), despite they need to interact tightly in their current configuration (depicted by the orange line). That way, and in an ideal hardware environment, we can deploy each Jadex Platform on a separate host, making the resulting Multipotent System the most flexible as possible. As we see in our descriptions in Section 3.4 focusing the evaluation of the concepts from our Multipotent Systems reference architecture, we currently need to step back from this idealized setting and make an appropriate compromise when deploying Multipotent System to real hardware.

3.3.2.2 Individualizing Jadex Platforms

We perform the concrete starting of Micro-Agents representing the different roles from the prototypical implementation of the Multipotent System reference architecture with an additional Micro-Agent called *StarterMicroAgent* dedicated to only this duty (cf. Listing 3.3). That way, we can ensure that all Micro-Agents that run on a specific Jadex Platform are started in the correct order we require. We need to do that because, as we stated in Section 3.3.1.3, all services a Micro-Agent provides are available first after finishing its initialization state within its life cycle. Further, different Micro-Agents may take different time to set up their functionality during their initialization state (cf. Section 3.3.1). Moreover, some Micro-Agents may require to access the provided services of other Micro-Agents during their initialization routine. Thus, we require to start one Micro-Agent after the other in the specified order (cf. the *List<Tuple2<String, CreationInfo>> ast* in Listing 3.3) when setting up a specific Jadex Platform.

Therefore, we start the *StarterMicroAgent* in the *AbstractPlatformStarter* (cf. Listing 3.1, Line 25) and include the information in the list of Micro-Agents to start on the respective Jadex Platform (cf. *asts* entered in the so-called Micro-Agent's *CreationInfo* in Listing 3.1, Line 23). The *StarterMicroAgent* then reads this information in its initialization state (cf.

Listing 3.3, line 19) and starts processing the information in its lifetime state (cf. Listing 3.1, Line 28). Subsequently, the *StarterMicroAgent* calls the implementation of its only provided service *SIMicroAgentStarted* internally (cf. Listing 3.3, Line 24). This starts the first Micro-Agent from the list of Micro-Agents *asts* we require it to start (cf. Listing 3.1, lines 11 and 39) and increases its index for working through the list (cf. Listing 3.1, line 41).

Because all Micro-Agents in our prototypical implementation (including the *StarterMicroAgent*) extend our default implementation of a Micro-Agent (cf. *AbstractMicAgent* in Listing 3.4), we know that each Micro-Agent we start also calls the service *SIMicroAgentStarted* provided by the *StarterMicroAgent* when entering its lifetime state (cf. Listing 3.4, Line 22). That way, we can achieve that the *StarterMicroAgent* first starts the next Micro-Agent from its list *asts* (cf. Listing 3.4, Line 5) after the last Micro-Agent has finished its initialization routine, is now in its lifetime state, and all of its provided services are now available. The *StarterMicroAgent* then continues this procedure until it worked through the whole list of Micro-Agents it must start. Then, the *StarterMicroAgent* terminates itself by calling the respective service of the Jadex Platform's component management service with its own component identifier (cf. *cms.destroyComponent(agent.getComponentIdentifier())* in Line 39 of Listing 3.3.

By following that routine and by letting all implementations of a Multipotent-Agent's different roles extend our implementation of the *AbstractMicAgent* (cf. Listing 3.4), we can achieve that each Jadex Platform is set up and functional in the way we intend it to be for our Multipotent System reference architecture.

3.3.3 Realizing Agents of Multipotent Systems as Jadex Platforms

We instantiate a Multipotent-Agent from our reference architecture for Multipotent System by starting and launching a new Jadex Platform. On that Jadex Platform, we host the respective roles the Multipotent-Agent can adopt as described in Section 3.2, each as a Micro-Agent. That way, we can set up the Multipotent-Agent having all its relevant roles represented and available already shortly after initializing the Jadex Platform. While in Section 3.2, we describe that Multipotent-Agent first adopt roles when they become relevant while handling a SCORe mission, in our prototypical implementation, we let each Multipotent-Agent adopt all roles during its whole up-time and set them to be active or inactive respectively. We benefit from this technical decision as it schedules time that we necessarily require for starting the individual Micro-Agents to time of the Jadex Platform's initialization and avoids unnecessary time-outs during the productive time of an Multipotent-Agent. Because the possible roles a Multipotent-Agent α can adapt on SEMANTIC HARDWARE LAYER are only limited by the total number of possible capabilities in the Multipotent System, we make an exception for roles that are dependent on the Multipotent-Agent α's hardware configuration, i.e., the roles that are defined by the set of the Multipotent System's $\mathcal{SDH}_\alpha$. Instead of adopting all of the possible roles for CAPABILITY IMPLEMENTER (cf. Figure 3.3) every time, we let the SELF-AWARENESS PROVIDER on SEMANTIC HARDWARE LAYER start and stop the respective Micro-Agents representing those roles when it detects changes done to $\mathcal{SDH}_\alpha$.

To realize this self-awareness mechanism, we create an event-based algorithm we depict in the activity diagrams in Figure 3.25 and Figure 3.26. We need to handle the effect of connecting a new SDH to an Multipotent System α's configuration, i.e., extending $\mathcal{SDH}_\alpha$ which we depict in Figure 3.25, and the effect of removing an existing SDH from a Multipotent System α's configuration, i.e., reducing $\mathcal{SDH}_\alpha$ which we depict in Figure 3.26.

```
1  @jadex.micro.annotation.Agent
2  @ProvidedServices({@ProvidedService(name = "SIMicroAgentStarted", type =
   SIMicroAgentStarted.class, scope = Binding.SCOPE_PLATFORM)})
3  public class StarterMicroAgent extends AbstractMicroAgent implements
   SIMicroAgentStarted {
4    // the index of the next Micro-Agent to start
5    private int aTs = 0;
6    // all Micro-Agents to start on the Jadex Platform as a list
7    List<Tuple2<String, CreationInfo>> asts;
8    // Jadex internal service used to start Micro-Agents
9    private IComponentManagementService cms;
10
11   // the initialization routine of the Micro-Agents life cycle
12   @AgentCreated
13   public void init() {
14     // load list of Micro-Agents from the CreationInfo
15     asts = (List<Tuple2<String, CreationInfo>>) agent.getComponentFeature(
   IArgumentsResultsFeature.class).getArguments().get("agents");
16   }
17
18   // the lifetime routine of the Micro-Agent's life cycle
19   @AgentBody
20   public void agentBody() {
21     // initialize the ComponentManagementService to start Micro-Agents
22     cms = SServiceProvider.getService(agent.getExternalAccess(),
   IComponentManagementService.class, RequiredServiceInfo.SCOPE_PLATFORM).get();
23     // initialize the Micro-Agent start-up routine for this Platform
24     agentStarted();
25   }
26
27   // provided service implementation for SIMicroAgentStarted
28   @Override
29   public IFuture<Void> agentStarted() {
30     // start remaining Micro-Agents if any
31     if (asts.size() > aTs) {
32       cms.createComponent(asts.get(aTs).getFirstEntity(), asts.get(aTs).getSecondEntity());
33       // update index for next Micro-Agent to start
34       aTs += 1;
35     }
36     // shutdown the Micro-Agent otherwise
37     else {
38       // shut down the Micro-Agent using its unique component identifier
39       cms.destroyComponent(agent.getComponentIdentifier());
40     }
41     // answer the service call
42     return IFuture.DONE;
43   }
44
45   // the shutdown routine of the Micro-Agent's life cycle
46   @AgentKilled
47   private void terminate() { super.teminate(); }
48 }
```

Listing 3.3: *StarterMicroAgent*

Figure 3.23: A Micro-Agent we use for initializing each Jadex Platform individually, ensuring the correct start-up sequence for the list of Micro-Agents that should run on the platform.

```
1  @jadex.micro.annotation.Agent
2  public abstract class AbstractMicAgent implements ICommunicationFeatureProvider {
3    // interface for internal \microagent functionality
4    @jadex.micro.annotation.Agent
5    protected IInternalAccess agent;
6    // communication interface for the \microagent
7    private ICommunicationFeature communication;
8    // specify if {\microagent}s should be automatically started after
     init is done
9    protected boolean sendAgentStartedOnInit;
10
11   // default routine started in the \microagent's initialization state
12   @AgentCreated
13   public void init() {
14     // initialize the communication interface
15     setCommunication(new DefaultCommunicator(agent));
16   }
17
18   // default routine started in the \microagent's lifetime state
19   @AgentBody
20   public void executeBody() {
21     // call the service SIMicroAgentStarted on platform
22     communication.communicateAsynchronously().onPlatform().useServiceType(
     SIMicroAgentStarted.class).useSingleService(SIMicroAgentStarted::agentStarted);
23   }
24
25   // default routine started in the \microagent's termination state
26   @AgentKilled
27   public void terminate() {     // do nothing on default
28   }
29 }
```

Listing 3.4: *AbstratcMicAgent*

Figure 3.24: The default Micro-Agent implementation we extend for all other Micro-Agents from our prototypical implementation of the Multipotent System reference architecture.

When connecting a new SDH, e.g., SDH_x in Figure 3.25, to an Multipotent System, we exploit the fact that we realize each instance of an SDH as a Jadex Platform. Thus, we can let SDH_x actively inform the Multipotent System in its role of a SELF-AWARENESS PROVIDER (cf. *call service connectSdh* in Figure 3.25). The SELF-AWARENESS PROVIDER subsequently can analyze the self-description handed over by SDH_x for analyzing the potential new capabilities the Multipotent-Agent α gains by extending its $\mathcal{SDH}_\alpha$ with SDH_x and stores these in a set *newCaps*. Additionally, the SELF-AWARENESS PROVIDER analyzes the current set of capabilities $\mathcal{C}_\alpha$ the Multipotent-Agent α can provide before connecting SDH_x and stores these in *agentCaps*. For every capability c in *newCaps* that is not already also included in *agentCaps*, the SELF-AWARENESS PROVIDER then can instruct the Jadex Platform's component management service to create a new component, i.e., start the respective Micro-Agent. This Micro-Agent then acts as a representative for the Multipotent-Agent's newly gained capability and realizes the role of a CAPABILITY IMPLEMENTER OF c. The CAPABILITY IMPLEMENTER OF c then, on the one

hand, informs the SDH_x about its existence, i.e., registers itself as a new corresponding Micro-Agent for future executions of the capabilities that SDH_x offers to the Multipotent-Agent. On the other hand, the CAPABILITY IMPLEMENTER OF c informs the SELF-AWARENESS PROVIDER via the Jadex Platform's component management service about its successful initialization. The SELF-AWARENESS PROVIDER then can add c to the set of *agentCaps* and remove c from the set of *newCaps* (cf. the SELF-AWARENESS PROVIDER swimming lane in Figure 3.25). If there are additional capabilities that SDH_x can offer to the Multipotent-Agent, i.e., if the set *newCaps* is not yet empty, the SELF-AWARENESS PROVIDER repeats the sequence for starting a respective Micro-Agent via the Jadex Platform's component management service. When finally all capabilities that SDH_x offers are represented by respective Micro-Agents realizing the roles of CAPABILITY IMPLEMENTERS, the SELF-AWARENESS PROVIDER has to analyze whether there are additional virtual capabilities the Micro-Agent can provide according to the changes to its $\mathcal{C}_\alpha$. Therefore, the SELF-AWARENESS PROVIDER first creates a new set *virtualCaps* consisting of these virtual capabilities, re-initializes the set of *newCaps* with them, and restarts the procedure of starting respective Micro-Agents in the case that not all capabilities in *virtualCaps* are not yet included in the set of already provided $\mathcal{C}_\alpha$, i.e., in *agentCaps*. Otherwise, the SELF-AWARENESS PROVDIDER can finalize the activity of integrating the new SDH_x to $\mathcal{SDH}_\alpha$ of the Multipotent-Agent.

Removing an SDH from an Multipotent System α's configuration potentially reduces the set of capabilities $\mathcal{C}_\alpha$ that α can provide. This is the case if no other SDH is included in α's set $\mathcal{SDH}_\alpha$ that also provides the capability directly as a physical capability or indirectly in combination with other SDH, i.e., as a virtual capability. We depict the necessary procedure of disconnecting an exemplary SDH_x in Figure 3.26. The SDH_x actively triggers the process of deregistering an SDH by calling the respective *disconnectSDH()* service provided by the Micro-Agent realizing the role of the SELF-AWARENESS PROVIDER each Multipotent System α adopts. This Micro-Agent then analyzes the current configuration of α concerning SDH *connectedSDH()* and retrieves the set of capabilities that potentially become obsolete after disconnecting SDH_x. After removing SDH_x from $\mathcal{SDH}_\alpha$, the Micro-Agent realizing the role of the SELF-AWARENESS PROVIDER then analyzes for every single one of these capabilities whether there is any other SDH in the reduced set $\mathcal{SDH}_\alpha$ that also offers the capability. This can be the case, e.g., if in case c is $c^p_{\text{M-TEMP}}$ there were two SDH connected that both encapsulate a temperature sensor. Then, disconnecting SDH_x has no further influence on the set of capabilities $\mathcal{C}_\alpha$. The same holds if there is no other SDH offering the same capability c directly as a physical capability but can offer it in combination with other SDH as a virtual capability. In that case, we only need to adapt the SDH the CAPABILITY IMPLEMENTER OF c must address when executing c. We achieve this by letting the SELF-AWARENESS PROVIDER call the service *updateSdhToAddress()* provided by the Micro-Agent realizing α's role of a CAPABILITY IMPLEMENTER OF c with the respective set *c.shds* that consists of SDHs. The CAPABILITY IMPLEMENTER OF c then can process this information and inform all involved SDH (we depict this only in a simplified version in Figure 3.26, cf. $\forall s \in$ *c.sdh: register Micro-Agent at s* in the swim lane of CAPABILITY IMPLEMENTER OF c). If there is no SDH nor any combination of SDH that can offer the respective capability c, the SELF-AWARENESS PROVIDER needs to interact with the Jadex Platform's component management service for destroying the CAPABILITY IMPLEMENTER OF c (cf. service call *destroyComponent(c)* in Figure 3.26). The component management service then stops the respective Micro-Agent by directly calling the respective *terminate()* method. Therein, the Micro-Agent unregisters itself from SDH_x so that they can also finish its deregistration process at Micro-Agent α and is available for being connected to another Micro-Agent α' if

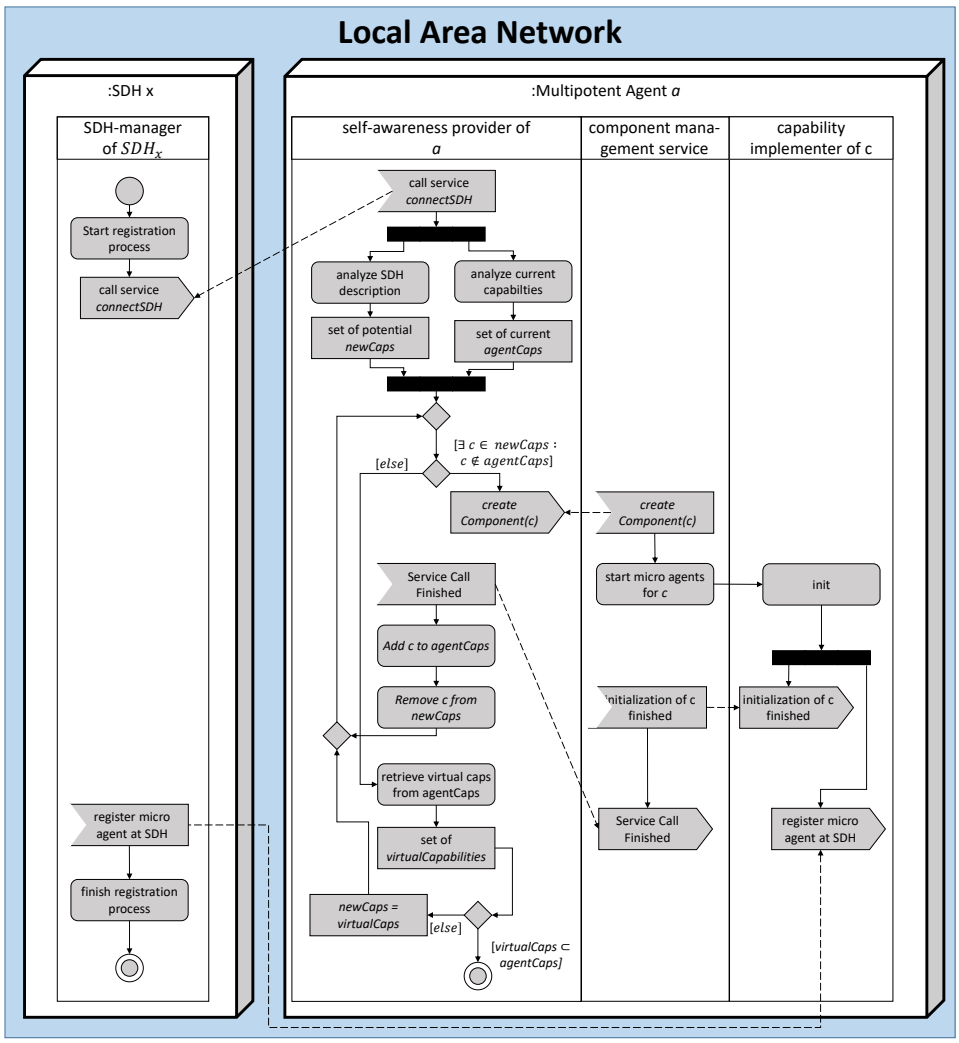


Figure 3.25: Activity diagram depicting how we realize the self-awareness ability of an Multipotent-Agent when registering new SDH to $\mathcal{SDH}_\alpha$. After starting the registration process of an SDH by calling the respective provided service offered by the Micro-Agent realizing the role of an SELF-AWARENESS PROVIDER, this Micro-Agent further interacts with the Jadex Platform's internal component management service for starting a new component if necessary. New components in the form of Micro-Agents realize possible new roles of CAPABILITY IMPLEMENTERs a Multipotent-Agent can additionally adopt in its new configuration.

required. In parallel, it can inform the component management service when it successfully executed the routines defined in its termination state. The component management service then forwards this information to the SELF-AWARENESS PROVIDER that can continue with the process for unregistrating SDH_x until all capabilities SDH_x was involved in are stopped for the Multipotent-Agent.

Using the described routines for registering and unregistering different SDH from the configurations of Multipotent-Agents enables the required flexibility concerning each Multipotent-Agent's set of available capabilities $\mathcal{C}_\alpha$.

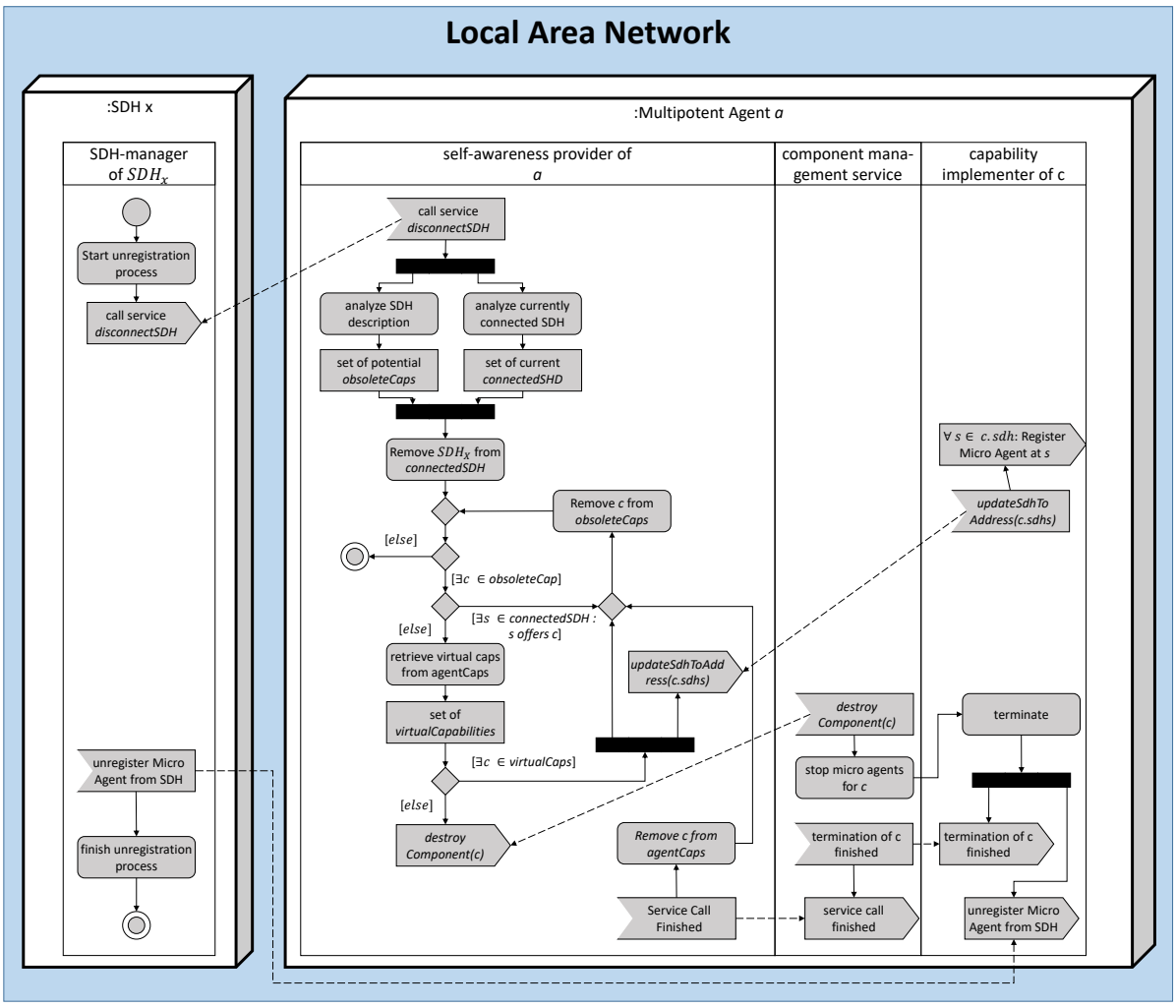


Figure 3.26: Activity diagram depicting how we realize the self-awareness ability during unregistering an exemplary SDH_x from α's set of $\mathcal{SDH}_\alpha$. After starting the process by calling the respective provided service offered by the Micro-Agent realizing the role of an SELF-AWARENESS PROVIDER, the SELF-AWARENESS PROVIDER analyzes whether the physical capabilities SDH_x can also be represented virtually by a respective combination of still existent other SDH from $\mathcal{SDH}_\alpha$ or not. Depending on the result of this analysis, the SELF-AWARENESS PROVIDER then either interacts with the Micro-Agents representing the respective CAPABILITY IMPLEMENTERS informing them to now register with other SDH for executing their capability or interacts with the Jadex Platform's internal component management service for destroying the respective components.

3.3.4 Realizing Self-Descriptive Hardware as Jadex Platforms

Besides Multipotent-Agents, we also realize all different instances of SDH as Jadex Platforms. We do this to create an adapter to real hardware that helps us abstracting from technical details if necessary. Concerning our Multipotent System reference architecture, we can interpret the Jadex Platform we create for each SDH-instances as an implementation of the semantic shell

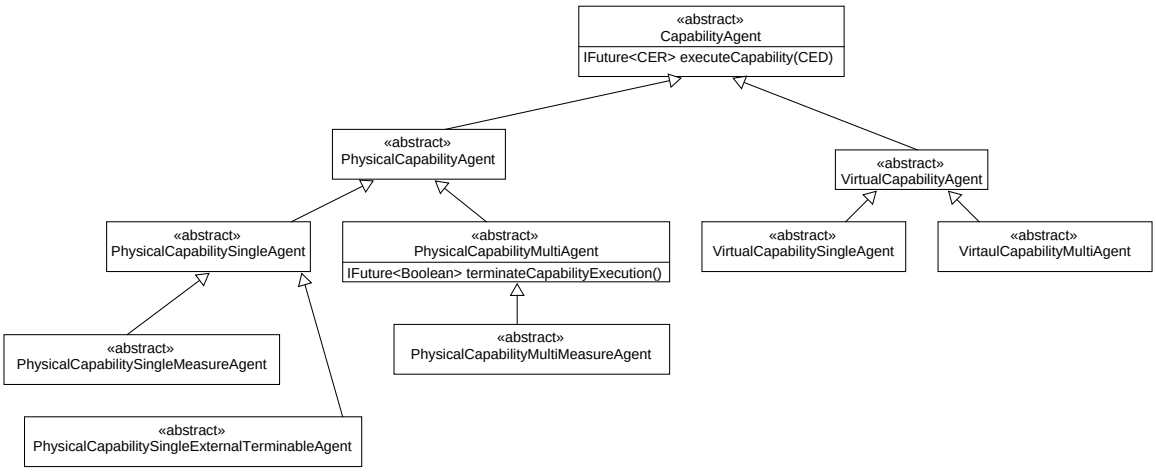


Figure 3.27: The Micro-Agent class hierarchy is abstractly defining the possible types of capability functionality provided by different SDH.

[Wanninger et al., 2018]. That way, each SDH can offer the standard interface to the rest of the system and provide the necessary self-description of hardware modules. This helps us with creating the connection between Multipotent-Agents and SDH during reconfigurations and with executing the different SDH's functionality.

By making all SDH active using the Jadex-concept of Micro-Agents, we can establish and dissolve connections between SDH and Multipotent-Agents, as we describe in Figure 3.25 and Figure 3.26. We, therefore, let every SDH be managed by an SDH MANAGER that serves as a relay between the encapsulated S&A and the surrounding Multipotent System. It holds the relevant information necessary for determining the set of available capabilities $\mathcal{C}_\alpha$ a Multipotent-Agent can provide in a specific configuration $\mathcal{SDH}_\alpha$. Thus, it can pass this information to the Micro-Agent realizing the role of a SELF-AWARENESS PROVIDER when the SDH gets connected to a Multipotent-Agent by the Multipotent System's user.

We structure this information into a class hierarchy, indicating which kind of capability the SDH can provide and thus which roles of CAPABILITY IMPLEMENTERS an Multipotent-Agent can adopt. We give an overview over the respective Micro-Agent class hierarchy we support in our prototypical reference architecture in Figure 3.27. To be conform with the Jadex-specific requirements, we design the abstract Micro-Agent realizing the general role of the CAPABILITY IMPLEMENTER as a *CapabilityAgent* and follow that naming convention for all its sub-types. Each *CapabilityAgent* provides the service *IFuture<CER extends CapabilityExecutionResult> executeCapability(param: CED extends CapabilityExecutionData)* on SEMANTIC HARDWARE LAYER to the Multipotent-Agent's role of a CAPABILITY COORDINATOR (cf. Figure 3.3), where *CER* (a sub-type of *CapabilityExecutionResult*) is a generic place holder for the concrete data type wrapping the *result* from a specific capability's execution and *CED* (a sub-type of *CapabilityExecutionData*) is a generic place holder for the concrete data type wrapping the parameters *param* for the specific capability's execution. Both *CER* and *CED* get further specified by the concrete type of *CapabilityAgent*. To express the different concepts of physical capabilities and virtual capabilities, we separate Micro-Agents extending the *CapabilityAgent* into those that are *PhysicalCapabilityAgent*s and those that are *VirtualCapabilityAgent*s. To

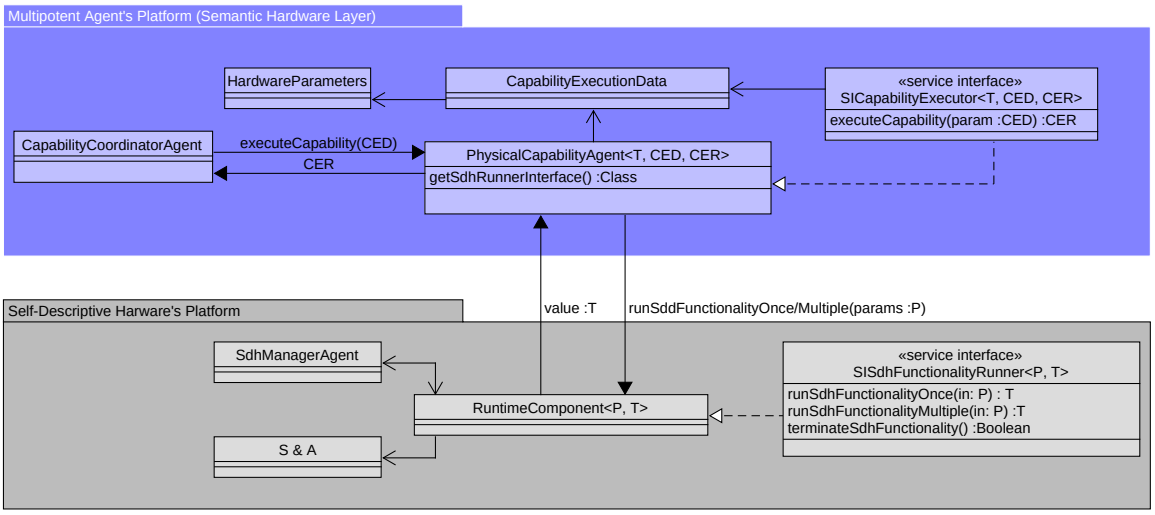


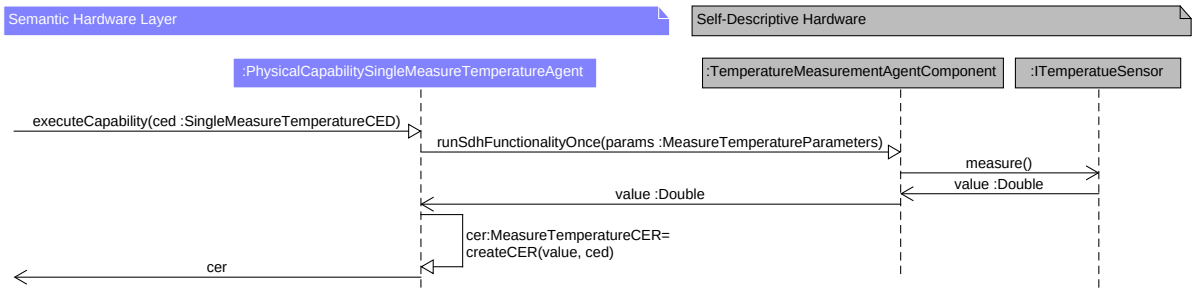
Figure 3.28: Sketch of an exemplary interaction and data types we use when Micro-Agents running on a Multipotent-Agent's Jadex Platform and an SDH's Jadex Platform cooperatively execute a physical capability. Arrows with filled tips indicate Jadex communication between the Micro-Agents, arrows with single lines indicate data type associations, and arrows with blank tips implementations of service interfaces.

indicate whether the capability produces a single result or multiple results when executed, we further differentiate the type of CAPABILITY IMPLEMENTER for physical capabilities as well as for virtual capabilities into *SingleAgent*s and *MultiAgent*s. Besides different *CER*, *SingleAgent*s and *MultiAgent*s also encapsulate a different logic for implementing the execution of their capability. To stop the stream of results the *MultiAgent*s thus provide an additional service *IFuture<Boolean> terminateCapabilityExecution()*. We further integrate a sub-type *PhyisicalCapabilityExternalTerminableAgent* of the *CapabilityAgent* for representing those capabilities that cannot terminate on their own but require an external coordination signal, e.g., for moving with a given velocity $c^{p}_{\text{MV-VEL}}$ (cf. Figure 3.29c). These also provide the additional service *IFuture<Boolean> terminateCapabilityExecution()* for stopping their execution when running. Because we assume that the set of different capabilities in a Multipotent System gets extended especially concerning its physical measuring capabilities, we further include the concepts of a *PhysicalCapabilitySingleMeasureAgent* and a *PhysicalCapabilityMultiMeasureAgent* realizing an abstract measuring capability in our hierarchy, easing future extensions of our prototypical implementation (cf. Figure 3.27). By structuring the information concerning the capabilities an SDH can provide that way, the SELF-AWARENESS PROVIDER can launch the correct type of CAPABILITY IMPLEMENTER after a certain SDH got connected to a Multipotent-Agent's configuration.

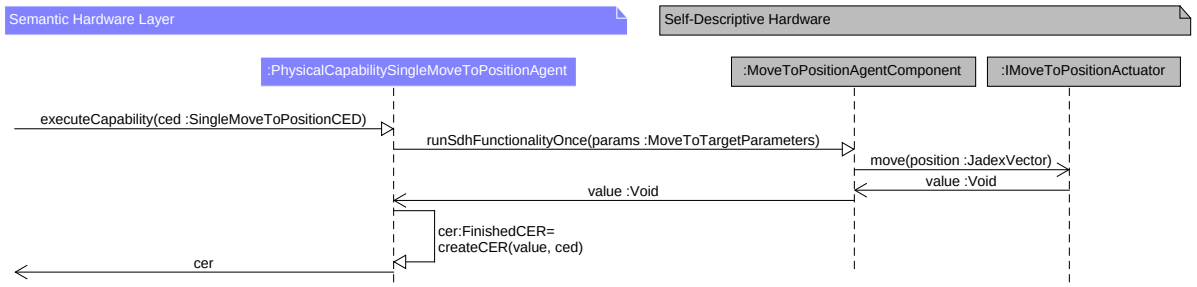
After this connection is established, the Micro-Agent realizing the SDH MANAGER running on the SDH's Jadex Platform also delivers the required interface for executing the SDH's functionality to the respective Micro-Agent that realizes the concrete CAPABILITY IMPLEMENTER running on the Multipotent-Agent's Jadex Platform. It does this in the common Jadex-conform way for communicating between Micro-Agents in the form of a provided service. To increase modularity and reuse of concepts, we separate the different functionality an SDH can have in dif-

ferent *RuntimeComponents* (cf. Figure 3.28). That way, we can use implementations of certain *RuntimeComponents* for different similar SDH. Specific sub-types of that *RuntimeComponents* then provide the respective services to the CAPABILITY IMPLEMENTER via the communication interface of the SDH MANAGER's Micro-Agent. This means, depending on their concrete type, different SDH MANAGER provide the respective relevant services *runSdhFunctionalityOnce() :T*, *runSdhFunctionalityMultiple() :T*, and *terminateCapabilityExecution() :Boolean* implemented in associated *RuntimeComponent*s, where T is the generic return type of the value returned by the function. Thereby, the SDH MANAGER can also control the resources of the S&A by scheduling incoming service calls, e.g., necessary to handle multiple access to the hardware. For clarity purposes in Figure 3.28, we avoid this separation into multiple sub-types and sub-interfaces. Instead, we give a sketch subsuming those into one concept (*RuntimeComponent<P,T>*) and one interface (*SISdhFunctionalityrunner<P,T>*). In this description, P is the generic type of the respective *RuntimeComponent*'s required *HardwareParameters* necessary for executing its functionality. The actual functionality, i.e., the hardware driver, then is implemented in an associated class dedicated to the specific S&A of the SDH. Because the concrete *RuntimeComponent* is designed for that specific S&A, it contains the logic for transforming the parameters P for actually accessing the S&A and for transforming its resulting value back to T that then again can be interpreted by the rest of the Multipotent System.

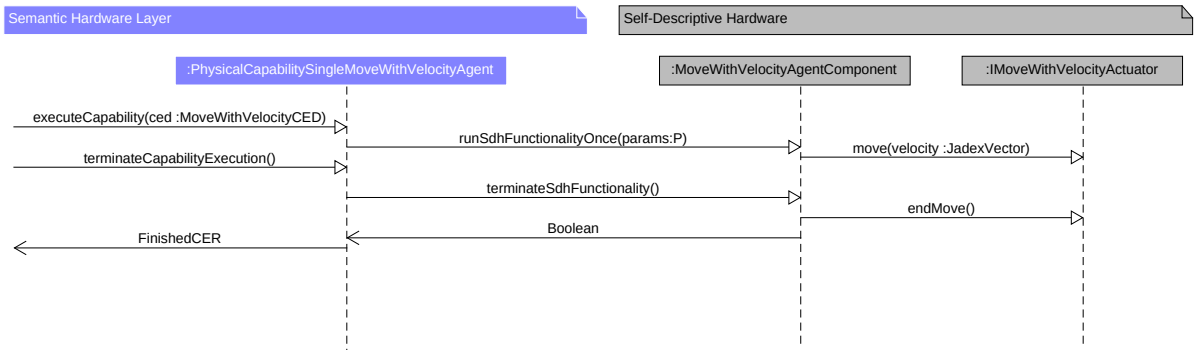
In Figure 3.29 we depict the concrete execution of different concrete types of capabilities as sequence diagrams. We start describing the sequence after an initial call of the provided service of the respective Micro-Agents that realizes the role of CAPABILITY IMPLEMENTERs of the Multipotent-Agent. Figure 3.29a describes the sequence of executing the capability $c^{p}_{\text{M-TEMP}}$. The respective Micro-Agent here is the *PhysicalCapabilitySingleMeasureTemperatureAgent* that requires a *SingleMeasureTemperatureCED* containing the relevant parameters *MeasureTemperatureParameters* for triggering the provided service of the SDH MANAGER's *AgentComponent* from the connected SDH that encapsulates a temperature sensor as S&A (cf. Figure 3.28). With that information, the *TemperatureMeasurementAgentComponent* then can execute the *measure()* method of the *ITemperatureSensor* interface implemented by the temperature sensor. The resulting *value* of the measurement, a variable of the type *Double*, can be returned to the CAPABILITY IMPLEMENTER OF $c^{p}_{\text{M-TEMP}}$. To be conform with the defined interface between SDH and the Multipotent-Agent, the *PhysicalCapabilitySingleMeasureTemperatureAgent* then encapsulates *value* in a *MeasureTemperatureCER* that can be interpreted by the rest of the Multipotent System and sends it back to the initiating CAPABILITY COORDINATOR. The sequence for other *PhysicalCapabilitySingleAgent*s only slightly differs from this procedure. Obviously, we require other parameters for the capability's execution, i.e., a position to move to in the form of a Jadex-conform *JadexVector*, which we encapsulate in a respective *SingleMoveToPositionCED* we send from the CAPABILITY COORDINATOR to the CAPABILITY IMPLEMENTER OF $c^{p}_{\text{MV-POS}}$. When executing $c^{p}_{\text{MV-POS}}$ which we describe in Figure 3.29b, e.g., we do not require to return a value instantly for further interpretation. Instead, we require it to first return after actually finishing the execution, i.e., when the Multipotent-Agent got positioned at the commanded position by the SDH (e.g., by an SDH encapsulating an UAV). That way, the CAPABILITY COORDINATOR can be sure about the Multipotent-Agent's state which it requires to act conform to the task of a plan it currently participates in its role of a PLAN WORKER. While $c^{p}_{\text{M-TEMP}}$ and $c^{p}_{\text{MV-POS}}$ both can terminate their execution on their own, we also support such capabilities that have no possibility for doing so. In Figure 3.29c, we describe the execution of $c^{p}_{\text{MV-VEL}}$ that is of such a kind. When receiving the initial service call from the CAPABILITY COORDINATOR the respective CAPABILITY IMPLEMENTER OF $c^{p}_{\text{MV-VEL}}$, i.e., the



(a) Executing $c^{p}_{\text{M-TEMP}}$ with a *PhysicalCapabilitySingleMeasureTemperatureAgent* realizing the role of the CAPABILITY IMPLEMENTER OF $c^{p}_{\text{M-TEMP}}$. Returns a measured Double encapsulated in a *TemperatureCER* to the calling CAPABILITY COORDINATOR after measuring the value.



(b) Executing $c^{p}_{\text{MV-POS}}$ with a *PhysicalCapabilitySingleMoveToPositionAgent* realizing the role of the CAPABILITY IMPLEMENTER OF $c^{p}_{\text{MV-POS}}$. Returns a Void encapsulated in a *FinishedCER* to the calling CAPABILITY COORDINATOR after reaching the commanded position.



(c) Executing $c^{p}_{\text{MV-VEL}}$ with a *PhysicalCapabilitySingleMoveWithVelocityAgent* realizing the role of the CAPABILITY IMPLEMENTER OF $c^{p}_{\text{MV-VEL}}$. Returns a Void encapsulated in a *FinishedCER* to the calling CAPABILITY COORDINATOR after stopping the movement due to a previous external call of *terminateCapabilityExecution()*.

Figure 3.29: Sequence diagrams describing the procedure of executing different physical capabilities triggered by the Multipotent-Agent in its role of a CAPABILITY COORDINATOR and managed by the respective CAPABILITY IMPLEMENTERS that address concretely instantiated *RuntimeAgentComponents* supplied by the SDH MANAGERS running on the respective SDH's Jadex Platform.

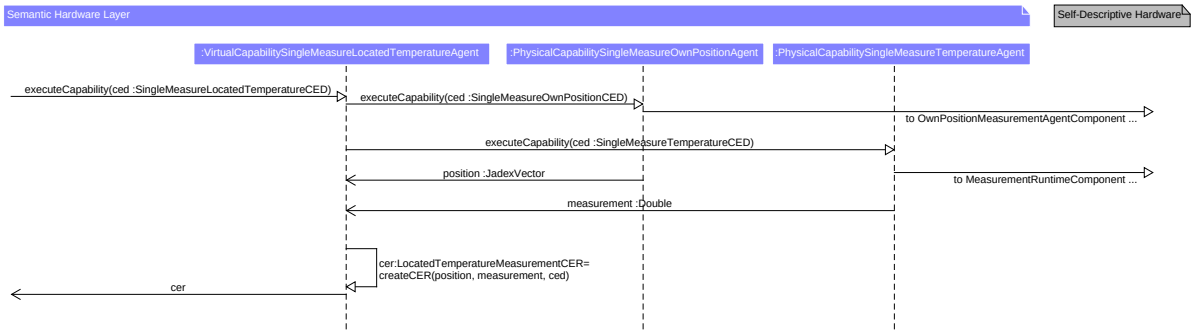


Figure 3.30: Sequence diagram describing the procedure of executing a virtual capability triggered by the Multipotent-Agent in its role of a CAPABILITY COORDINATOR and managed by the respective CAPABILITY IMPLEMENTERs that address concretely instantiated *RuntimeAgentComponents* supplied by the SDH MANAGERs running on the respective SDH's Jadex Platform.

Micro-Agent *PhysicalCapabilitySingleMoveWithVelocityAgent*, can only start the execution of the capability but cannot finish it without further external control. After starting the SDH's functionality by executing *runSdhFunctionalityOnce* that the SDH MANAGER forwards via its *MoveWithVelocityAgentComponent* to the implementation of the respective *IMoveWithVelocityActuator*, moving with the commanded *velocity* can only be terminated by calling *endMove()*. The required external control signal then can only originate from the CAPABILITY COORDINATOR that forwards it to the CAPABILITY IMPLEMENTER OF $c^{p}_{\text{MV-VEL}}$ who then can finish the capability's execution. Although it is not depicted in Figure 3.29c, we can assume that the coordination signal stems from either the PLAN COORDINATOR the involved Multipotent-Agent works within its role of a PLAN WORKER or directly from the Multipotent System's user.

In Figure 3.30, we describe the general execution of virtual capabilities for located measurements in our prototypical implementation of the Multipotent System reference architecture. Executing a virtual capability can involve multiple other capabilities. Generating a located measurement requires two capabilities to be executed, i.e., $c^{p}_{\text{M-POS}}$ for determining the position of the measurement as well as the actual measuring capability. In our prototypical Jadex implementation, we realize virtual capabilities as Micro-Agents as we do it with other capabilities. That way, we can give a general and abstract implementation of virtual capabilities for different located measurements as we depict in Figure 3.30. When any concrete implementation of the abstract Micro-Agent *VirtualCapabilitySingleMeasureLocatedAgent* (cf. Figure 3.28) receives a call of its provided service *executeCapability*, this call includes the relevant parameters for all involved physical capabilities. The *VirtualCapabilitySingleMeasureLocatedAgent* then forwards the included concrete *CED* to the respective Micro-Agents realizing the involved CAPABILITY IMPLEMENTERs. Because in any of its concrete implementations the *VirtualCapabilitySingleMeasureLocatedAgent* requires a position measurement, it calls *executeCapability* with a *SingleMeasureOwnPositonCED* provided by the *PhysicalCapabilitySingleMeasureOwnPositionAgent*. After receiving the result of this call, i.e., the current *position* encoded in a *JadexVector*, and after receiving the *measuredValue* of the respective concrete *PhysicalCapabiltiySingleMeasureAgent* that is also involved in the virtual capability, the *VirtualCapabilitySingleMeasureLocatedAgent* can combine these results in a *LocatedMeasureCER* which it then returns to the initiating CAPABILITY COORDINATOR that can further process it as required within its task. In Figure 3.30, we depict the sequence executed when calling the concrete im-

plementation of a *VirtualCapabilitySingleMeasureLocatedTemperatureAgent* that realizes the virtual capability of a located temperature measurement. It generates

- a position measurement by calling the provided service *exeuteCapability* of the *PhysicalCapabilitySingleMeasureOwnPositionAgent*, i.e., the Micro-Agent realizing a Multipotent-Agent's role of a CAPABILITY IMPLEMENTER OF $c^{p}_{\text{M-POS}}$, and
- a temperature measurement by calling the provided service *executeCapability* of the *PhysicalCapabilitySingleMeasureTemperatureAgent*, i.e., the Micro-Agent realizing the Multipotent-Agent's role of a CAPABILITY IMPLEMENTER OF $c^{p}_{\text{M-TEMP}}$ (cf. Figure 3.29a), and then
- combines these to a *LocatedTemperatureMeasurementCER* that it returns to the CAPABILITY COORDINATOR that initiated the located measurement.

By following this design scheme, the Multipotent-Agent can execute all of its provided capabilities in its different roles of CAPABILITY IMPLEMENTERS, each realized with a respective Micro-Agent on the Multipotent-Agent's Jadex Platform. That way, we can hide the technical details for accessing the encapsulated S&A for all roles a Multipotent-Agent adopts on the different layers of the Multipotent System reference architecture.

3.3.5 Realizing the User's Device as Jadex Platform

Like every instance of a Multipotent-Agent and that of SDHs, we realize the user's device as a Jadex Platform using the same start-up procedure as we describe it in Section 3.3.2.1. For instructing and supervising the Multipotent System during real-world executions as well as when running in simulation only, we design the user's device with a graphical front-end (cf. Figures 3.31 to 3.33 and 3.34a to 3.34c). In this front-end, we can display different information concerning the system in multiple tabs. We created the user's device to start one Micro-Agent for each of these tabs, providing services and requesting its dedicated duty. This way, the user's device is flexibly extendable to new Micro-Agents collecting, aggregating, and displaying new information from the system or interacting with the Multipotent System as a whole or single components of it, i.e., individual Multipotent-Agents or SDH-instances. In the following, we give a brief overview of the current possibilities for controlling and interacting with the Multipotent System and collecting information about it in our prototypical reference.

3.3.5.1 The HTN-Design View

In a tab dedicated to the design of new SCORe-mission-specific HTN, the user can create, save, load, and modify HTNs. That way, we provide the possibility to the user for

- designing completely new HTN using the provided control elements for creating Complex-NodesCNs, defining conditions for their decomposition into Primitive-NodesPNs, modifying the world state WS, and explicitly triggering re-plannnig Replanning-Node (RP)s (cf. left-hand side of Figure 3.31).
- store so created HTN to a library of valid HTN snippets and load already created HTN-snippets from that library, enabling the user to combine HTN-snippets to realize even more complex SCORe missions (cf. bottom button line of Figure 3.31).
- visualize and debug the decomposition of created HTN to avoid inconsistencies and programming failures as early as possible (cf. right-hand side of Figure 3.31).

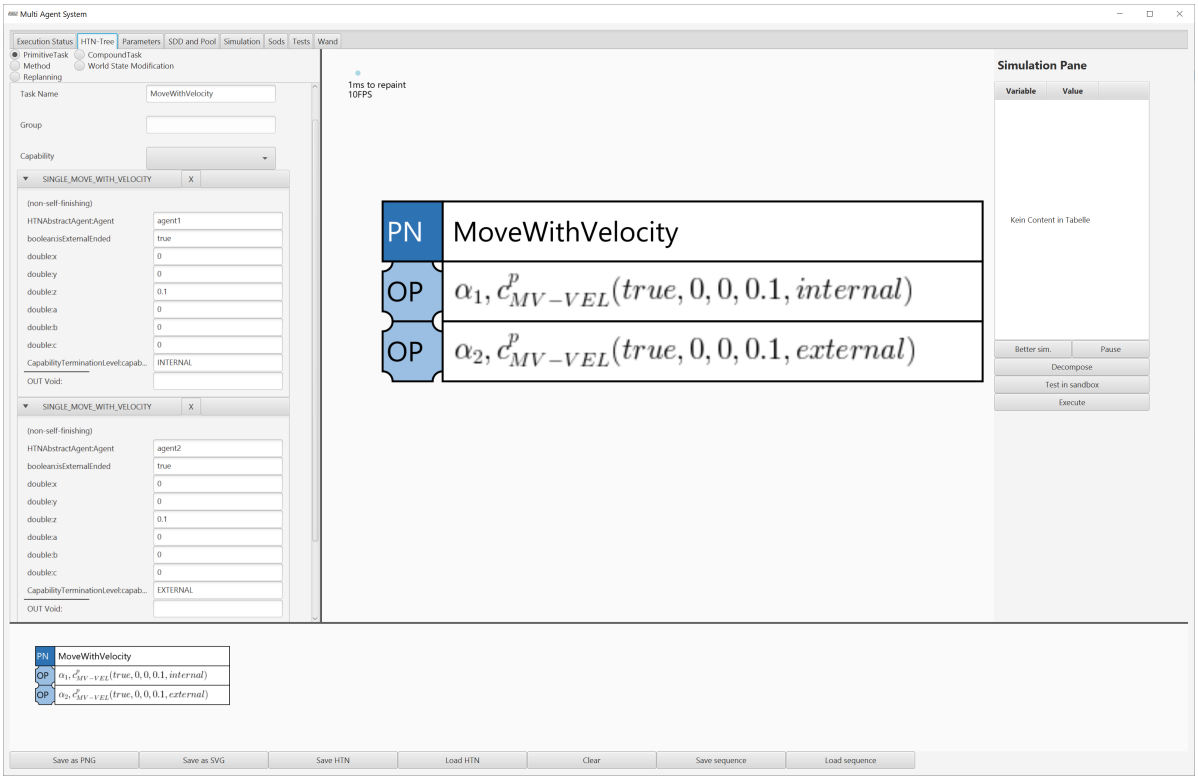


Figure 3.31: Activated tab in the graphical front-end providing access to the Multipotent System running on the user's device providing the necessary control elements for designing SCORe mission-specific HTN.

- create a plan using the designed HTN and world state and let the Multipotent System instance try to execute this plan.

To provide access to other parts of the Multipotent System, we start a Micro-Agent on the Jadex Platform we set up for the user's device. This Micro-Agent takes the user inputs from the HTN-design view and directs them to the Multipotent-Agents for their further processing, i.e., lets the Multipotent System execute created plans.

3.3.5.2 The Agents View

In a tab dedicated to the different Multipotent-Agents in the running Multipotent System, we can display up-to-date information about each of these Multipotent-Agents. These contain

- information on the currently active Multipotent-Agents, identified by their Jadex-specific unique identifier (left-hand side of Figure 3.32). The user can select each of the displayed entries to view further information concerning the respective Multipotent-Agent.
- information concerning the selected Multipotent-Agent's current configuration, i.e., which concrete SDH-instances are connected to the Multipotent-Agent, including data on the resulting capabilities this configuration, provides to the Multipotent-Agent (cf. right-hand side of Figure 3.32). The user can switch between different views on this information, grouping information either for capabilities (and thus display the respective SDH they

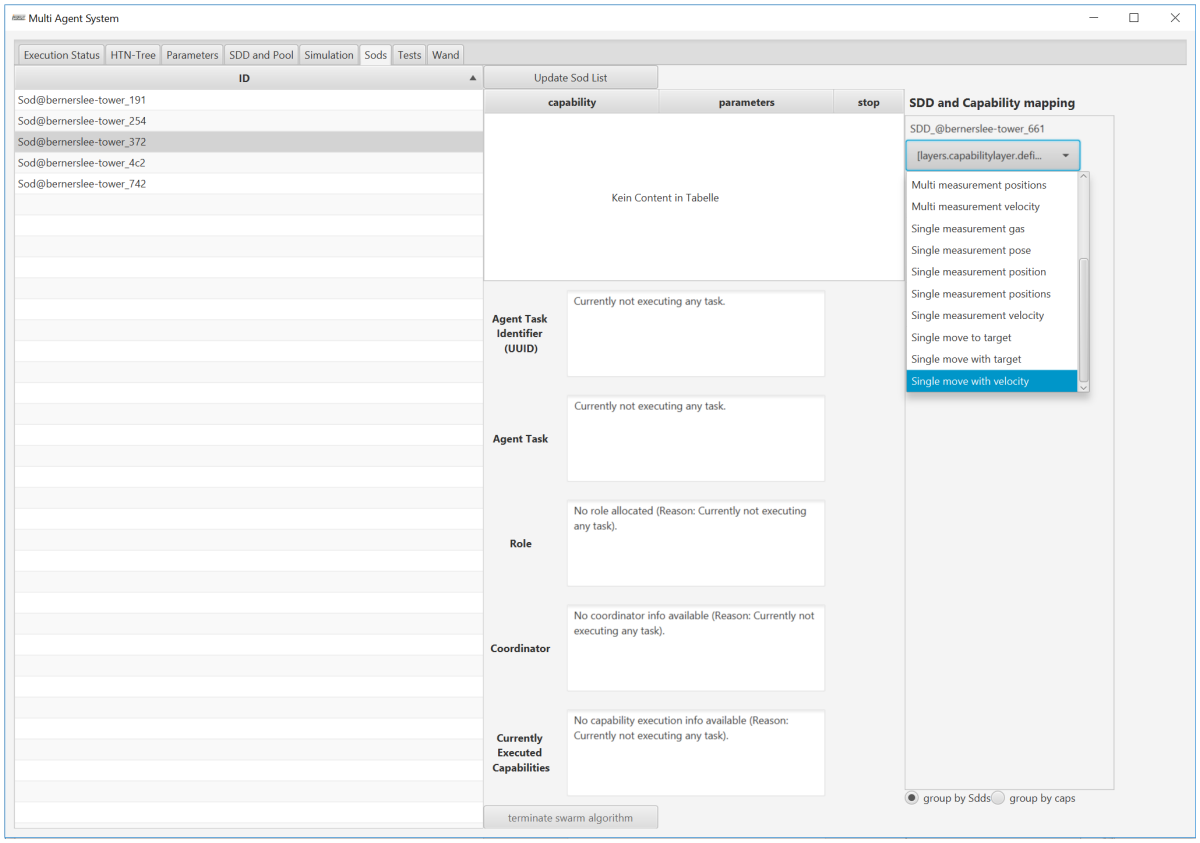


Figure 3.32: Activated tab in the graphical front-end providing access to the Multipotent System, displaying information concerning the different active Multipotent-Agents and their $\mathcal{SDH}_\alpha$ and $\mathcal{C}_\alpha$ (Multipotent-Agents are called Sod in the Jadex implementation).

originate from) or grouping information for SDH (and thus display the respective capabilities that originate from that SDH).

- information concerning the selected Multipotent-Agent's roles within a plan (whether it is a PLAN COORDINATOR or PLAN WORKER), the respective task in that plan, and the capabilities it currently executes within that plan, if it has any task assigned and is not idle (cf. the middle column of Figure 3.32).

To receive updates and display changes in the Multipotent System's internal states, we start a Micro-Agent on the Jadex Platform for the user's device. Each Multipotent-Agent actively communicates with when its status changes.

3.3.5.3 The SDH-Instances View

In a tab dedicated to the different SDH-instances in the Multipotent System, we can give the user an overview of the details of each of these SDH-instances.

- Active SDH are separated into three different views (identified by Jadex-internal identifiers), indicating whether the SDH is currently not connected to any specific device (can only be the case in simulated environments), is connected to a Multipotent-Agent, or is

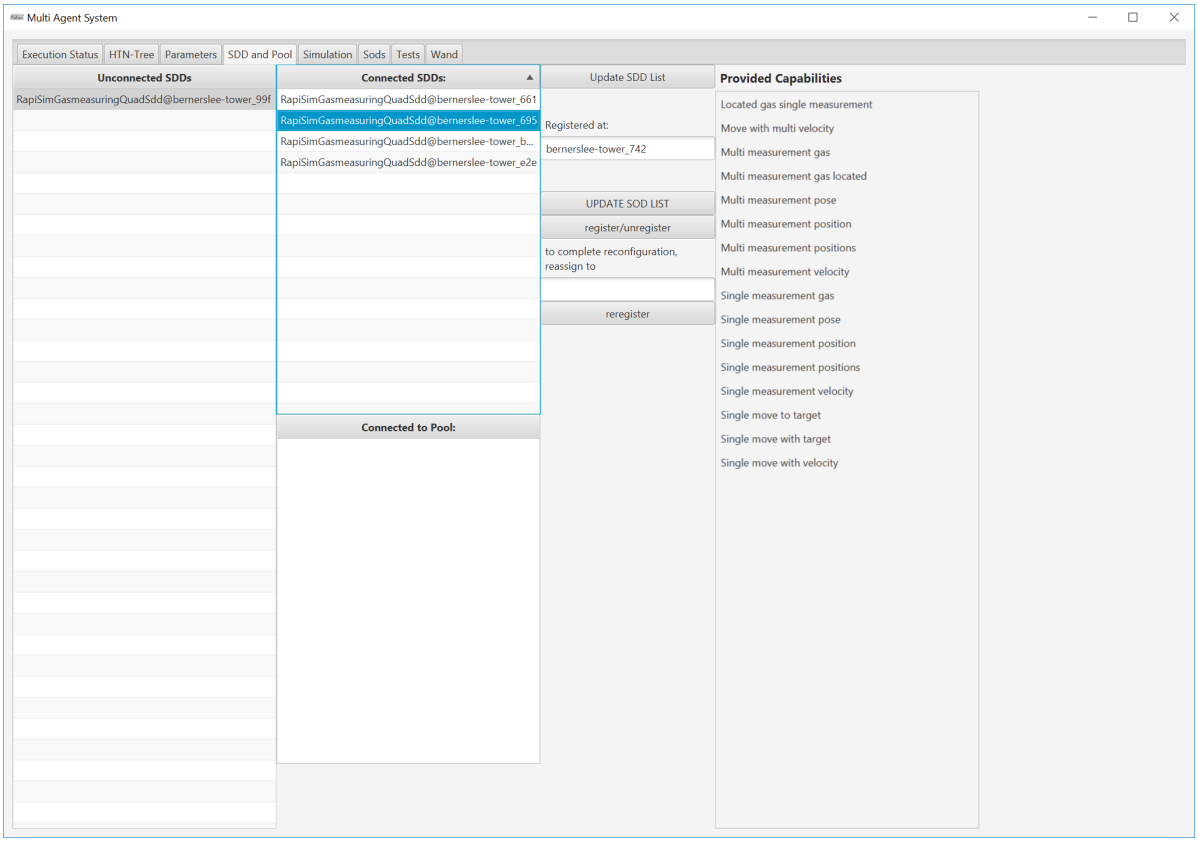


Figure 3.33: Activated tab in the graphical front-end providing access to an information on the SDH-instances running in the Multipotent System (SDH are called SDD in the Jadex implementation).

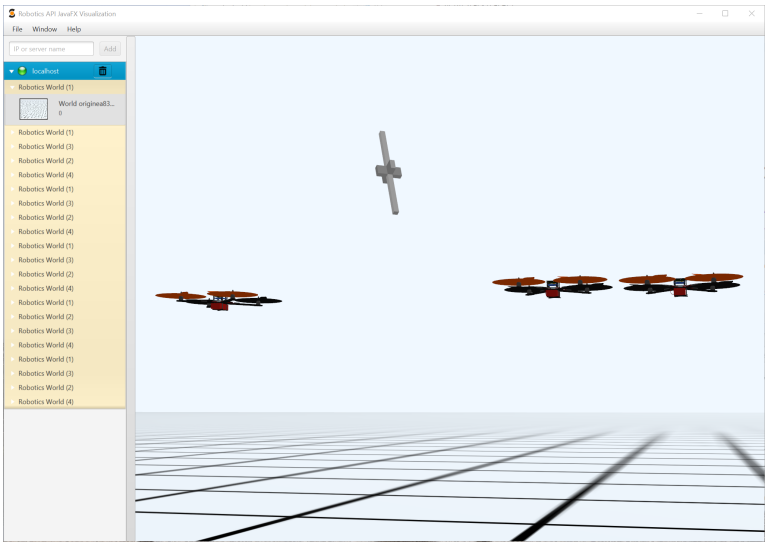
connected to a pool of additional SDH connected to the user's device (cf. left-hand side of Figure 3.33).

- By selecting an SDH in any view, we provide control elements for connecting and disconnecting SDH to other devices and display information further describing the current state of the SDH, i.e., which physical capabilities the SDH can offer to a Multipotent-Agent and to which Multipotent-Agent the SDH currently is connected to (cf. right-hand side of Figure 3.33).

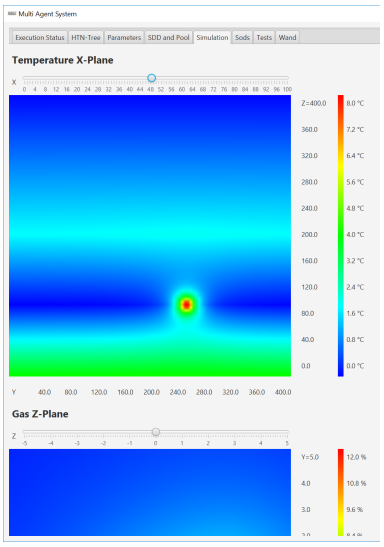
To receive updates and display changes in the SDH-instances' internal states, we start a Micro-Agent on the Jadex Platform for the user's device that each SDH-instance actively communicates with when its status changes. Further, this Micro-Agent can interact with each SDH-instance to perform the connecting and disconnecting required during reconfiguration when we run the Multipotent System in a simulated environment.

3.3.5.4 The Simulation Environment Views

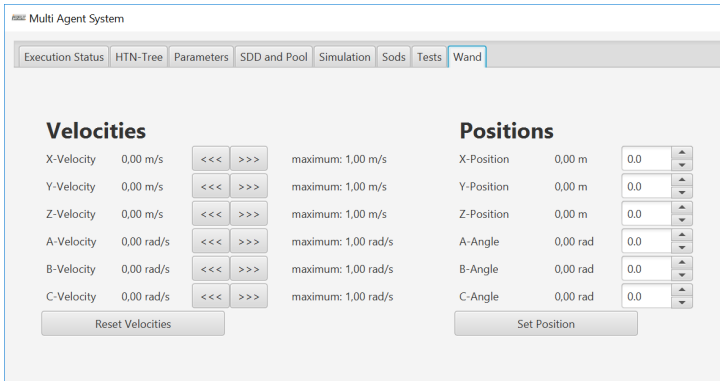
When running a Multipotent System in simulation, we provide controlling possibilities and different views on the simulated entities to the user, which we depict in Figure 3.34.



(a) The RAPI-based [Angerer et al., 2013] visualization for showing moving entities like SDH-instances encapsulating UAV.



(b) The visualization of static simulated environment parameters.



(c) The graphical control interface for steering the pointing device.

Figure 3.34: Graphical front-end providing access to a simulated Multipotent System instance.

- We provide a RAPI-based [Angerer et al., 2013] visualization of the moving entities in our simulation environment which we depict in Figure 3.34a. There, we can display those SDH-instances encapsulating UAV (cf. the three UAV depicted in Figure 3.34a) and other simulated moving devices that the user can control (cf. the pointing device in Figure 3.34a).
- We provide the possibility to control the pointing device visualized in Figure 3.34a with an input panel we depict in Figure 3.34c. By pressing the respective buttons, the user can control the pointing device in the simulation environment in all 6-dimensions (position and rotation) with direct feedback in the visualization depicted in Figure 3.34a.
- We provide a visualization for simulated environment parameters (currently only with static behavior) we can measure with a simulated SDH-instance. We depict the additional tab of our graphical front-end in Figure 3.34b. The depicted state exemplary displays

temperature and gas distribution in the simulated environment.

Again, for each of these views we use for interacting with simulated Multipotent System instances, we start a Micro-Agent on the Jadex Platform of the user's device, providing the necessary interaction interface to the user on the one side and providing appropriate communication interfaces to the other Micro-Agents running on other Jadex Platforms respectively.

3.4 Evaluation

We evaluated the feasibility of implementing our reference architecture for Multipotent System in different prototypical versions for flying ensembles. For realizing these prototypes, we used the Jadex framework as described in Section 3.3. Our evaluation concerning the prototypical implementation of the Multipotent Systems reference architecture demonstrates

- the feasibility of executing SCORe missions in the real world, involving flying ensembles constructed with real hardware that implement the concepts of the Multipotent Systems reference architecture as we describe them in Section 3.3,
- the appropriate interplay of different mechanisms and technologies of the Multipotent System reference architecture we introduce in Section 3.2, and
- the practicability of physical reconfigurations concerning the physical hardware configuration $\mathcal{SDH}_\alpha$ of agents.

In the following, we present our results in these three areas and thus demonstrate the general feasibility of using Multipotent System to accomplish real-world SCORe missions. Detailed evaluations concerning the core technologies of Multipotent System, i.e., our approach for *Ensemble Programming for Multipotent Systems*, our approach for *A Self-Organization Mechanism for Ensemble Formation*, our approach for *A Self-Organization Mechanism for Physical Reconfiguration*, and our approach for *Executing Ensemble Programs by Using Self-Organization* including the *execution of Collective Capabilities encapsulating swarm behavior* are then evaluated in the respective Chapters 4 to 7.

Evaluation Setting for This Section We settled the evaluations we performed in this section primarily in the context of Environmental Monitoring. There, we identified that the investigation in conditions of the Atmospheric Boundary Layer (ABL) that geographic researchers regularly perform [Wolf et al., 2017] could be formulated as an instance of a SCORe mission. Often, certain conditions in the ABL have to be continuously observed, e.g., to enrich meteorological climate models with appropriate real-world data (cf. our Case Study on *Improving Climate Models* introduced in Section 2.3). Therefore, our goal during two real-world field experiments we performed during the Environmental Monitoring campaigns ScaleX 2015 [Wolf et al., 2017] and ScaleX 2016 [Kosak et al., 2018; KIT IMK/IFU, 2018; ISSELabs, 2018] was

- to verify the feasibility of deploying our respective prototypical implementations of the reference architecture for Multipotent System to flying ensembles build with real hardware,
- to demonstrate that we can use these flying ensembles for executing SCORe missions in the context of Environmental Monitoring, and
- to show that flying ensembles have the potential to be used for collecting valuable data for detecting and analyzing meteorological phenomena like temperature inversion.

As we were focusing on the feasibility of accomplishing a SCORe mission in general, we performed all experiments during the environmental measuring campaigns ScaleX 2015 and ScaleX 2016 with a fixed set of capabilities $\mathcal{SDH}_\alpha$ for all involved agents. We neglected the ability to reconfigure their physical configurations. To also evaluate the feasibility of the physical reconfiguration of agents during run-time, we designed another laboratory experiment. With that experiment, we were able to demonstrate that

- creating Self-Descriptive Hardware (SDH) encapsulating real S&A is feasible,
- we can configure agents with such real hardware-based SDH-prototypes at run-time, and
- an agent's set of capabilities $\mathcal{C}_\alpha$ actually can be modified that way.

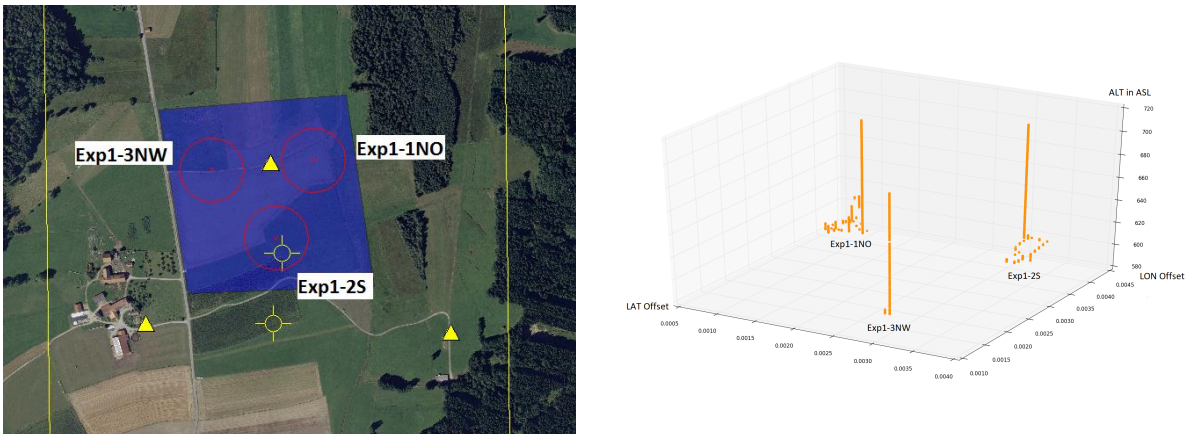
We achieved such with multiple different SDH-prototypes, focusing on such providing distance information.

3.4.1 Field Experiment ScaleX 2015: Deploying Flying Ensembles to SCORe Missions for Environmental Monitoring Using Real Hardware

Throughout our first field experiment during the ScaleX 2015 environmental measuring campaign [Wolf et al., 2017], we focused on demonstrating the feasibility of deploying real hardware to SCORe missions in general and validate that flying ensembles can derive valuable insights in the case study's setting. Therefore, we aimed at generating useful measurements in-situ achieved by a flying ensemble. The experiment was carried out at the Fendt, Peißenberg, Germany, 47.827600 N, 11.059959 E, Experimental Site (DE-Fen) on June 30th of 2015.

3.4.1.1 Experiment Design

In the course of our ScaleX 2015 experiment, a plan contained a coordinated flight pattern for the synchronous ascend of an ensemble from $0m$, i.e., ground level, up to $100m$ AGL and a subsequent synchronous descent of the ensemble back to $0m$ AGL. We designed the plan for an ensemble consisting of three agents providing the capabilities $c^p_{\text{MV-POS}}$, $c^p_{\text{M-POS}}$, $c^p_{\text{M-TEMP}}$, and $c^p_{\text{M-HUM}}$ each. Further, we required the ensemble to perform the flight pattern (synchronous ascend and descend) in three different positions (Exp1-1NO, Exp1-2S, and Exp1-3NW in Figure 3.35a). During this flight, we required each agent within the ensemble to measure time-synchronized vertical profiles of temperature and humidity gradients, i.e., execute their capabilities $c^p_{\text{M-TEMP}}$ and $c^p_{\text{M-HUM}}$. For a subsequent evaluation of the derived measurements, we further required each agent to combine each measurement of $c^p_{\text{M-TEMP}}$ and $c^p_{\text{M-HUM}}$ with the respective position the measurement was performed. We achieved this by letting each agent execute additional position measurements derived by executing $c^p_{\text{M-POS}}$. For illustration purposes, Figure 3.35b depicts the position measurements derived by the three agents of the ensemble executing $c^p_{\text{M-POS}}$ during one exemplary execution of the plan. Thus, the plan in ScaleX 2015 resulted in a very similar plan for the ensemble like the one from the running example we introduced in Section 3.2 (i.e., a part of $\rho_{\text{NBL-S}}$, performed with three instead of only one agent). The full mission in ScaleX 2015 consisted of the frequent execution of this plan. During a full day, we required our ensemble to work through that plan at each full hour. This resulted in 24 plan executions in total, each producing a time-synchronous measurement profile of the temperature and humidity gradient from the three positions mentioned above. During the experiment in ScaleX 2015, the plan for the whole flying ensemble was generated using the QGroundControl mission planning software



(a) ScaleX 2015 mission design, map data © 2021 GeoBasis-DE/BKG (© 2009)

(b) ScaleX 2015 flight pattern

Figure 3.35: Experimental setup of ScaleX 2015 illustrating the SCORe mission's flight pattern performed by the ensemble consisting of three agents. Figure 3.35a shows the triangular sensor drone formation experiment setup at DE-Fen on June 30 of 2015. Figure 3.35b illustrates the flight pattern performed by the ensemble in every execution of a plan within the SCORe mission by depicting the position measurements derived by an ensemble consisting of three agents executing $c^{p}_{\text{M-POS}}$ during one exemplary flight.

[Dronecode, 2019] for UAV. Our goals within this experiment and its SCORe mission were to detect the occurrence of a temperature inversion in the Nocturnal Boundary Layer (NBL) and investigate changes in humidity gradients while doing that. This way, we could demonstrate the feasibility of deploying flying ensembles implementing the Multipotent System reference architecture for executing real-world SCORe missions and produce and evaluate relevant data for the case study of *Improving Climate Models* from the research field of Environmental Monitoring we introduce in Section 2.3. Moreover, data collectively retrieved by the flying ensemble we deployed as an innovative measuring instrument can be used for verifying state-of-the-art remote sensing devices like the virtual 3D Doppler boundary layer lidar measuring tower (cf. our case study *Innovative Measurement Methods* in Section 2.4).

3.4.1.2 Software and Hardware Prototypes

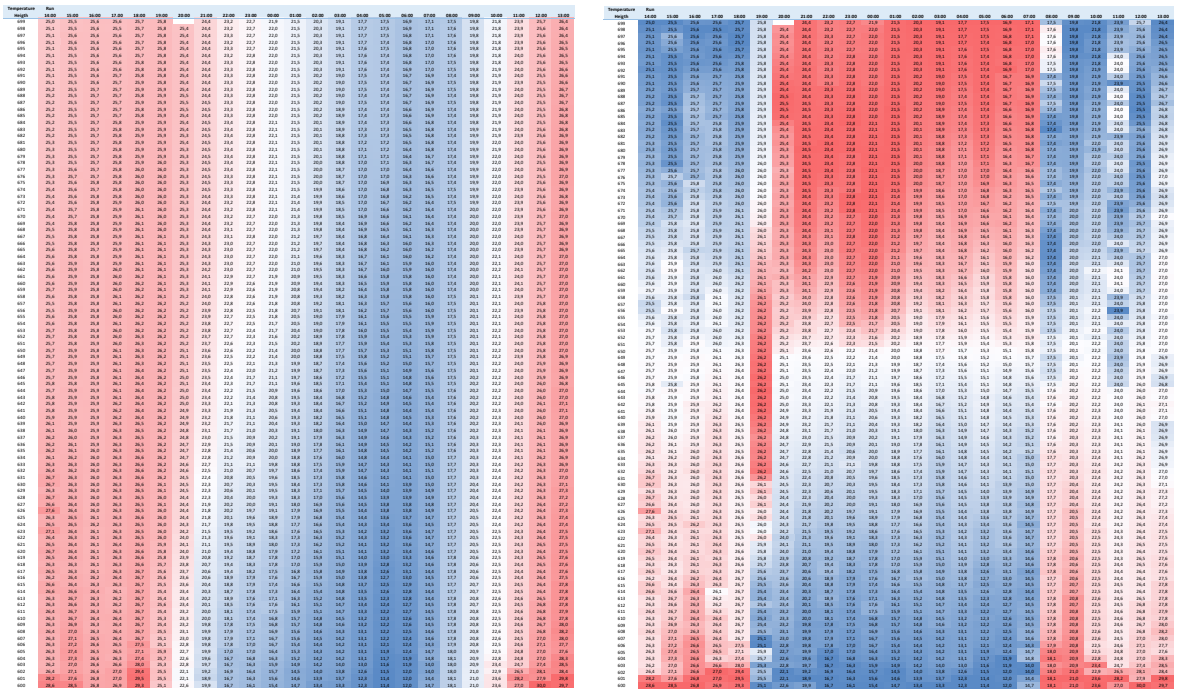
During the experiment, we used multiple semi-autonomous agents equipped with two SDH-prototypes each (cf. Figure 3.36). One of the SDH-prototypes ($\text{SDH}_{\text{SHT75}}$) encapsulated two SHT-75 sensors [Sensirion, 2018] that can create combined temperature and humidity measurements each. We designed the $\text{SDH}_{\text{SHT75}}$-prototypes with this redundancy to reduce possible inaccuracies that can occur when using lightweight temperature and humidity sensors [Wolf et al., 2017]. Carrying an $\text{SDH}_{\text{SHT75}}$-prototype enabled each agent to execute the capabilities $c^{p}_{\text{M-TEMP}}$ for measuring temperature and $c^{p}_{\text{M-HUM}}$ for measuring humidity. Another prototypical SDH we equipped each agent with ($\text{SDH}_{\text{AQ-S}}$) encapsulated an Autoquad flight controller [Autoquad, 2018] based UAV (cf. Figure 3.36), enabling each agent to execute $c^{p}_{\text{MV-POS}}$ for moving to a given position and $c^{p}_{\text{M-POS}}$ for measuring its position. We deployed the required software necessary for each agent on a portable single-board computer (Odroid XU3 [Hardkernel, 2018]),

Figure 3.36: The ScaleX 2015 agent prototype, consisting of a an Odroid XU3 single board computer [Hardkernel, 2018] providing the run-time environment for the agents, an $\text{SDH}_{\text{SHT75}}$-prototype encapsulating a redundant SHT-75 sensor (combined temperature and humidity sensor) [Sensirion, 2018], and an $\text{SDH}_{\text{AQ-S}}$ prototype encapsulating an Autoquad flight controller based UAV [Autoquad, 2018], ©ISSE.

running a Linux-based OS for hosting the run-time environment for the agents. For each agent, the SDH prototypes $\text{SDH}_{\text{SHT75}}$ and $\text{SDH}_{\text{AQ-S}}$ were directly connected to the Odroid XU3 with fixed wiring. We used an nRF-antenna [NORDIC SEMICONDUCTORS, 2018] for communication purposes. In our ScaleX 2015 experiment, the configuration of agents concerning their $\mathcal{SDH}_\alpha$ was not yet reconfigurable, and the self-description of the $\text{SDH}_{\text{SHT75}}$-prototypes and the $\text{SDH}_{\text{AQ-S}}$-prototypes was hard-coded into the agent's software. Further, we performed most of the necessary coordination of the ensemble manually, i.e., Ensemble Formation, synchronization, and re-planning were human-controlled. Nevertheless, each agent was able to execute the necessary capabilities $c^p_{\text{M-TEMP}}$, $c^p_{\text{M-HUM}}$, and $c^p_{\text{M-POS}}$ required in the respective SCORe mission autonomously. Thus, also in this very early prototypical state of the architecture for Multipotent Systems, we were already able to demonstrate the feasibility of deploying real hardware-enabled, flying ensembles in real-world SCORe missions and generate useful application-specific data.

3.4.1.3 Experiment Execution and Results

In Figures 3.37a, 3.37b and 3.38, we give a full overview of all 24 plan executions performed during the SCORe mission of the ScaleX 2015 experiment, focusing on different aspects each. Results in each figure stem from the 24 plan executions performed during a full day in June 2015, starting on June 30th at 14:00 local time (left) and ending on July 1st at 13:00 local time (right). Measurements during these plan executions performed by agents in-situ were made every 0.5 seconds (s) in degrees Celsius ($°C$). In each figure, we depict the average



(a) overall flights colors gradients (b) flight-individual colors gradients

Figure 3.37: Results from the ScaleX 2015 experiment focusing temperature measurements. Individual entries in the matrix show the average temperature calculated from in-situ measurements performed by the flying ensemble in three different positions (cf. Figure 3.35). Colors indicate the gradient of temperature measurements for each measurement flight (data misses because of data inconsistencies for run number 7 at 20:00 at height 699m ASL). Measurement flights were repeated each full hour during a day, resulting in 24 runs in total. Measurement depicted start at a height of 600m ASL and range up to 699m ASL.

of measurements performed by the three agents for each height in between 600m ASL and 699m ASL while executing the synchronous flight pattern. For calculating this average value, all measurements derived at the respective height (rounded to full meters) were taken into account, i.e., for a given height in a specific run, we respect all measurements performed by any SHT-75 sensor encapsulated in any agent's $\text{SDH}_{\text{SHT75}}$-prototype within the ensemble that was derived at the same height. In Figures 3.37a, 3.37b and 3.38, we use relative color gradients ranging from *deep blue* (low measured value) over *white* to *deep red* (high measured value) to illustrate the changes in the respective measured value.

In Figure 3.37, we analyze temperature measurements derived during the experiment. We colorize the average temperature measurements according to the color gradient from deep blue for the lowest measured temperature to deep red for the highest measured temperature. While in Figure 3.37a, we perform this colorization for *the whole mission*, i.e., every 24 runs of the mission, we perform the colorization flight-individual in Figure 3.37b. When comparing the gradients for every run individually in Figure 3.37a, we can remark that the trend in the color gradient from intensive color at a lower height to less intense at higher heights stays the same for almost the whole day with a change from red at the day to blue at night. This indicates

Humidity Heigth	Run 14:00	15:00	16:00	17:00	18:00	19:00	20:00	21:00	22:00	23:00	00:00	01:00	02:00	03:00	04:00	05:00	06:00	07:00	08:00	09:00	10:00	11:00	12:00	13:00
699	36,6	37,3	37,8	36,1	33,0	31,9		40,5	46,0	49,0	50,9	53,5	59,4	63,3	70,2	70,7	73,5	71,7	70,8	63,9	56,5	55,0	44,8	37,9
698	36,6	37,4	37,5	35,5	33,0	31,9	36,0	40,5	46,0	49,0	50,9	53,4	59,1	63,3	70,4	70,7	73,7	71,8	69,9	64,3	56,7	55,0	45,2	37,8
697	36,6	37,6	37,5	35,3	33,0	31,9	35,9	40,5	46,0	49,0	51,0	53,4	59,1	63,1	70,4	70,5	73,7	71,7	70,3	64,6	56,8	54,9	45,2	37,8
696	36,6	37,6	37,6	35,4	33,0	31,8	35,9	40,5	46,0	48,9	50,9	53,4	59,1	63,2	70,5	70,7	73,6	71,8	70,5	64,7	56,9	54,9	45,2	37,7
695	36,6	37,5	37,4	35,6	33,1	31,8	35,9	40,5	46,0	48,9	50,9	53,4	59,1	63,4	70,6	70,7	74,0	71,8	70,0	64,6	57,0	54,8	45,3	37,7
694	36,6	37,4	37,4	35,7	33,2	31,8	35,8	40,4	46,0	48,9	50,9	53,4	59,1	63,3	70,6	70,8	73,9	71,9	70,2	64,5	57,1	54,8	45,4	37,6
693	36,6	37,4	37,4	35,7	33,3	31,8	35,8	40,4	45,9	48,9	50,9	53,4	59,2	63,4	70,9	70,4	73,9	72,0	70,7	64,4	57,1	54,8	45,4	37,6
692	36,6	37,3	37,2	35,5	33,2	31,8	35,8	40,4	45,9	48,9	50,9	53,4	59,1	63,6	70,8	70,8	72,9	72,1	70,7	64,5	57,0	54,8	45,3	37,7
691	36,5	37,4	37,2	35,4	33,3	31,7	35,7	40,4	45,9	48,9	50,9	53,3	59,1	63,6	71,1	70,9	74,4	72,1	71,8	64,4	57,1	54,8	45,2	37,8
690	36,5	37,4	37,3	35,3	33,3	31,7	35,8	40,3	45,9	48,9	50,9	53,4	59,2	63,7	70,9	71,2	74,2	72,2	70,9	64,5	57,1	55,0	45,2	37,8
689	36,5	37,4	37,2	35,3	33,2	31,7	35,7	40,4	46,1	48,9	50,8	53,4	59,1	63,7	71,4	71,1	74,2	72,2	71,6	64,4	57,1	54,7	45,1	37,9
688	36,5	37,3	37,3	35,3	33,2	31,7	35,8	40,4	46,1	48,8	50,8	53,4	59,2	63,8	71,6	71,7	74,1	72,3	71,9	64,7	57,3	54,6	45,2	37,9
687	36,5	37,2	37,4	35,1	33,1	31,8	35,8	40,4	46,1	48,9	50,9	53,4	59,3	64,0	71,2	71,5	74,4	72,0	71,7	64,8	57,0	54,6	45,0	37,7
686	36,4	37,2	37,4	35,1	33,0	31,7	35,8	40,3	46,1	48,9	50,8	53,3	59,3	64,0	71,6	71,6	74,7	72,3	72,0	64,9	57,2	54,5	45,1	37,7
685	36,4	37,4	37,4	35,2	33,0	31,7	35,9	40,3	46,1	48,8	50,8	53,4	59,4	64,1	71,8	71,9	75,4	72,2	72,6	64,8	57,4	54,4	45,0	37,6
684	36,4	37,4	37,4	34,9	32,9	31,7	36,1	40,3	46,1	48,8	50,8	53,4	59,5	64,3	71,7	72,1	75,1	72,5	72,9	64,8	57,6	54,5	45,0	37,6
683	36,4	37,4	37,3	34,9	32,9	31,7	36,2	40,3	46,2	48,9	50,8	53,5	59,5	64,3	71,8	72,1	75,6	72,4	73,3	64,8	58,0	54,4	44,8	37,6
682	36,4	37,4	37,3	34,9	32,9	31,7	36,3	40,3	46,2	48,9	50,8	53,5	59,6	64,5	72,4	72,4	75,6	72,7	72,6	64,9	57,8	54,5	45,0	37,6
681	36,4	37,3	37,3	34,9	32,9	31,7	36,4	40,3	46,2	48,8	50,8	53,5	59,6	64,4	72,7	72,7	75,8	73,0	72,6	65,0	57,8	54,4	45,0	37,6
680	36,4	37,3	37,3	35,1	33,0	31,7	36,5	40,3	46,2	48,8	50,8	53,6	59,7	64,8	72,9	72,5	76,2	72,7	72,3	64,8	58,2	54,4	45,0	37,6
679	36,4	37,3	37,3	35,2	32,9	31,7	36,6	40,2	46,2	49,0	50,9	53,6	59,7	64,9	72,9	72,8	76,3	73,4	72,5	64,5	57,9	54,4	44,9	37,6
678	36,3	37,2	37,2	35,2	33,0	31,7	36,6	40,3	46,3	48,9	50,8	53,7	59,7	64,9	73,8	73,2	76,3	73,2	72,4	64,6	58,1	54,3	44,9	37,6
677	36,3	37,3	37,3	35,0	33,2	31,7	36,8	40,2	46,3	49,0	50,8	53,7	60,0	65,2	73,5	73,6	76,1	73,6	72,0	64,8	58,1	54,3	44,9	37,6
676	36,3	37,3	37,3	35,2	33,2	31,7	36,8	40,1	46,4	49,0	50,9	53,9	60,1	65,4	73,6	73,5	76,8	74,1	72,0	64,8	58,1	54,2	44,9	37,7
675	36,3	37,3	37,1	35,3	33,0	31,7	36,8	40,2	46,3	49,0	50,9	53,9	59,9	65,2	73,8	73,8	76,5	74,2	72,4	65,1	58,2	54,1	44,8	37,6
674	36,3	37,3	37,2	35,4	33,1	31,7	36,8	40,1	46,4	49,1	50,9	54,0	60,2	65,6	74,1	74,5	76,9	74,4	72,0	65,2	58,3	54,1	44,9	37,7
673	36,3	37,2	37,1	35,4	33,0	31,6	36,9	40,1	46,3	48,6	50,9	54,2	60,4	65,6	74,1	74,7	77,0	74,6	72,6	64,9	58,5	53,9	44,9	37,7
672	36,3	37,2	37,1	35,2	33,1	31,6	36,9	40,2	46,4	48,6	51,0	54,3	60,6	65,9	74,1	74,5	77,1	75,1	72,2	65,4	58,5	53,8	44,8	37,7
671	36,2	37,3	37,2	35,2	33,1	31,6	36,9	40,0	46,4	48,6	51,0	54,5	60,6	66,0	74,5	75,2	77,0	75,2	72,8	65,1	58,4	53,8	44,6	37,7
670	36,2	37,3	37,1	35,1	33,1	31,6	36,9	40,0	46,5	49,2	51,1	54,7	61,0	66,2	74,7	75,6	77,2	75,3	72,7	65,1	58,4	53,8	44,5	37,7
669	36,2	37,2	37,1	35,1	33,1	31,6	36,9	40,0	46,4	49,3	51,1	54,7	60,9	66,1	74,7	75,4	76,9	75,6	73,1	64,9	58,1	53,8	45,0	37,8
668	36,2	37,1	37,1	35,0	33,1	31,6	37,0	40,0	46,5	49,3	51,1	54,9	60,9	66,4	75,1	75,8	77,5	75,7	73,4	65,0	58,1	53,8	45,0	37,7
667	36,1	37,0	37,0	34,8	33,0	31,6	37,0	40,1	46,5	49,4	51,2	55,0	61,2	66,4	75,8	75,8	77,6	76,1	73,9	65,2	58,0	53,8	45,0	37,7
666	36,2	37,0	37,0	34,9	33,0	31,6	37,0	40,0	46,6	49,5	51,3	55,2	61,5	66,9	75,4	76,3	77,9	76,3	74,3	65,1	57,8	53,8	44,9	37,6
665	36,3	36,9	36,9	34,9	32,9	31,5	37,1	40,1	46,6	49,4	51,4	55,2	61,7	66,8	75,9	76,6	77,8	76,9	74,4	65,7	57,7	53,9	44,8	37,5
664	36,3	36,9	36,9	34,9	33,0	31,5	37,1	40,1	46,8	49,6	51,3	55,8	62,0	67,1	76,5	76,8	77,9	77,0	74,6	65,2	57,5	54,0	44,6	37,5
663	36,2	36,9	36,9	34,7	32,9	31,6	37,2	39,9	46,8	49,6	51,6	56,0	61,8	67,4	76,4	77,0	78,4	78,3	74,6	65,1	57,5	53,9	44,4	37,6
662	36,1	37,0	37,0	34,8	33,0	31,6	37,3	40,2	46,9	49,6	51,7	56,1	62,2	67,5	76,2	77,3	78,8	78,3	74,8	64,8	56,8	53,8	44,5	37,7
661	36,2	36,9	37,0	34,8	32,4	31,6	37,1	40,3	46,9	49,9	51,7	56,4	62,4	67,7	76,8	78,4	79,2	78,1	74,6	64,7	57,1	53,9	44,4	37,8
660	36,3	37,0	37,0	34,9	32,7	31,6	37,3	40,3	47,0	50,0	52,0	56,5	62,4	68,1	77,2	77,8	79,2	78,6	74,7	64,6	57,3	53,9	44,5	37,9
659	36,4	36,9	36,8	34,9	32,9	31,6	37,2	40,3	47,0	50,1	52,1	56,9	62,8	68,3	77,3	78,6	79,1	78,6	74,9	64,7	57,2	53,9	44,6	37,8
658	36,6	36,9	36,8	34,9	32,8	31,6	37,3	40,5	47,1	50,1	51,9	57,4	63,0	68,7	77,8	78,5	79,0	78,9	74,6	64,6	57,1	54,0	44,4	37,6
657	36,7	36,9	36,7	35,1	32,8	31,6	37,2	40,5	47,1	50,3	52,1	57,4	63,3	68,8	77,8	78,9	80,5	79,3	74,7	64,6	57,1	53,9	44,0	37,5
656	36,5	36,7	36,6	35,1	32,7	31,6	37,2	40,7	47,2	50,5	52,6	57,6	63,6	69,2	78,1	78,8	80,5	79,4	74,4	64,6	56,7	53,9	44,1	37,4
655	36,4	36,7	36,6	35,1	32,7	31,7	37,2	40,6	47,4	50,6	52,8	58,5	64,0	70,0	78,3	79,4	81,0	79,9	74,7	64,7	56,9	53,9	44,2	37,4
654	36,4	36,7	36,5	34,9	32,3	31,7	37,2	40,8	47,6	50,6	53,2	58,3	64,2	69,9	78,5	79,6	81,1	80,1	74,5	64,6	56,8	53,9	44,2	37,4
653	36,2	36,7	36,4	35,0	32,1	31,7	37,2	40,7	47,9	51,5	53,4	58,8	64,3	70,2	78,6	79,9	81,8	80,4	74,9	64,4	56,6	53,8	44,0	37,5
652	36,0	36,7	36,4	34,8	32,1	31,7	37,2	41,0	47,8	51,6	53,7	59,4	64,7	70,5	78,9	80,1	81,8	80,5	74,2	64,5	56,7	53,7	44,0	37,4
651	35,9	36,7	36,3	34,9	32,1	31,7	37,3	41,1	48,0	51,4	53,9	59,7	65,0	71,0	79,3	80,4	82,3	81,6	74,6	64,4	56,6	53,5	44,0	37,4
650	35,9	36,8	36,4	34,7	32,1	31,7	37,3	41,3	48,1	52,1	54,5	60,2	65,4	70,6	80,0	80,6	82,6	81,9	74,3	64,5	56,6	53,5	43,7	37,4
649	35,8	36,8	36,3	34,8	32,1	31,7	37,3	41,5	48,4	51,9	54,5	60,1	65,1	71,2	79,3	81,1	82,3	82,0	74,9	64,7	56,7	53,6	44,1	37,4
648	35,8	36,7	36,3	34,7	32,1	31,7	37,3	41,5	48,5	52,5	54,7	60,5	66,3	72,3	81,3	81,3	83,1	81,9	74,4	64,5	56,7	53,5	44,3	37,4
647	35,7	36,7	36,3	34,9	32,1	31,7	37,3	41,7	48,7	52,5	55,6	60,6	66,6	73,0	80,8	81,6	83,4	82,0	74,6	64,5	56,8	53,5	44,1	37,3
646	35,7	36,6	36,3	34,8	32,1	31,7	37,4	41,5	49,0	53,8	55,2	61,3	67,3	73,4	80,6	82,2	83,7	82,8	74,7	64,5	56,9	53,4	43,9	37,2
645	35,7	36,5	36,3	34,9	32,2	31,8	37,3	42,0	49,2	53,0	55,6	61,8	67,6	73,5	81,9	82,2	83,5	83,1	74,3	64,4	56,9	53,4	43,7	37,1
644	35,8	36,4	36,3	34,8	32,1	31,8	37,5	42,0	49,4	53,5	56,4	62,1	67,0	73,8	81,5	82,5	83,4	84,0	74,5	64,1	56,9	53,4	43,9	37,3
643	35,7	36,5	36,4	35,1	32,2	31,9	37,3	41,8	49,4	54,0	56,7	62,5	68,3	75,1	82,2	84,0	84,4	84,5	74,1	64,4	57,0	53,2	44,0	37,2
642	35,7	36,4	36,4	35,0	32,2	31,9	37,5	42,2	50,0	54,4	56,8	64,0	67,5	75,0	82,2	83,3	84,3	85,0	74,4	64,1	57,1	53,3	44,2	37,3
641	35,7	36,5	36,4	35,2	32,4	32,0	37,6	42,4	50,7	54,3	58,1	63,4	67,7	75,8	83,1	83,8	84,9	85,1	74,4	64,1	57,0	53,2	44,0	37,2
640	35,6	36,6	36,6	35,3	32,5	32,2	37,7	42,6	51,1	55,6	57,7	64,4	68,3	75,9	82,6	83,9	84,1	85,2	74,2	64,0	57,1	53,2	43,9	37,1
639	35,4	36,6	36,5	35,3	32,7	32,3	37,8	43,2	51,8	55,8	58,9	64,1	68,6	76,3	83,3	84,3	84,6	84,6	74,2	64,1	57,1	53,1	43,8	36,8
638	35,5	36,7	36,6	35,3	33,0	32,4	37,8	43,4	51,6	56,1	59,3	65,8	70,6	77,2	84,0	85,0	85,7	85,3	74,4	63,8	57,4	53,1	43,8	36,8
637	35,4	36,7	36,7	35,2	33,3	32,4	38,0	43,8	52,5	56,9	59,5	65,7	71,2	76,6	83,9	85,1	86,0	85,3	74,2	64,0	57,2	53,4	43,7	36,8
636	35,3	36,7	36,6	35,2	33,6	32,4	38,1	44,3	52,3	57,4	60,3	66,6	71,8	78,0	83,9	85,8	86,6	85,3	74,3	64,0	56,9	53,4	43,7	36,8
635	35,2	36,5	36,6	35,1	33,8	32,4	38,3	44,5	52,8	60,0	60,9	67,0	72,7	77,8	84,8	86,1	86,7	84,3	74,5	63,9	57,0	53,1	43,8	36,8
634	35,4	36,5	36,5	35,1	33,9	32,4	38,3	44,8	54,1	56,5	60,8	67,0	72,9	77,8	84,9	86,4	87,3	86,0	74,1	63,9	56,8	53,0	43,8	36,7
633	35,0	36,5	36,5	35,0	33,9	32,4	38,7	45,3	54,7	53,8	61,9	67,5	73,6	78,2	84,9	86,8	87,3	85,6	74,5	64,1	56,7	53,0	43,6	36,6
632	34,4	36,7	36,4	34,9	33,9	32,5	38,6	45,9	54,8	57,9	62,7	68,9	74,1	79,0	85,5	86,8	87,3	85,3	74,3	63,8	56,7	53,1	43,7	36,5
631	33,6	36,8	36,4	35,0	33,7	32,7	38,8	46,6	56,5	59,7	63,3	69,1	75,7	79,8	86,0	87,6	87,6	85,8	74,6	63,6	56,8	53,0	43,6	36,5
630	33,7	36,8	36,3	35,0	33,6	33,2	38,8	46,9	56,7	61,4	64,0	69,9	74,8	79,5	85,7	87,5	88,6	85,6	74,6	64,1	56,8	53,1	43,5	36,4
629	34,9	36,8	36,3	35,0	33,4	33,1	39,1	47,3	57,1	63,0	63,7	69,9	75,8	80,1	86,8	88,1	89,1	86,2	74,6	64,3	56,8	52,9	43,5	37,0
628	34,2	36,9	36,2	34,7	33,3	33,3	39,4	46,8	58,5	62,7	64,6	70,2	77,1	80,7	86,6	88,6	88,9	85,9	74,8	63,7	56,9	53,0	43,7	36,8
627	34,5	36,8	36,2	34,8	33,1	33,3	39,4	49,0	59,6	62,0	67,0	72,2	77,2	81,0	86,9	88,8	90,0	86,5	74,6	64,0	56,8	52,9	43,4	36,8
626	32,6	36,8	36,3	34,8	33,0	33,5	39,6	49,9	59,6	63,8	65,6	72,4	77,4	81,4	88,0	88,4	89,9	85,5	75,0	64,0	56,7	52,9	43,4	36,9
625	35,2	36,5	36,3	34,6	33,1	33,5	39,8	50,1	60,0	64,6	67,6	71,9	78,2	82,0	87,6	89,5	90,3	86,2	75,0	64,0	56,5	52,8	43,4	36,7
624	34,9	36,6	36,2	34,6	33,2	33,9	40,2	50,7	61,6	64,5	68,6	73,3	78,2	82,2	88,0	90,4	90,4	87,9	75,2	64,1	57,0	53,0	43,0	36,8
623	33,7	36,9	36,5	35,0	33,2	34,0	40,5	52,0	63,3	66,7	69,3	72,9	79,1	82,3	88,8	91,6	89,9	86,6	75,2	64,1	56,8	52,7	43,2	36,8
622	35,2	36,9	36,4	34,8	33,2	34,0	41,8	53,5	62,2	66,5	70,8	75,4	80,4	83,5	88,1	91,1	90,0	87,0	75,4	64,1	56,8	52,6	43,5	36,8
621	35,2	36,7	36,4	34,9	33,3	35,0	41,3	54,5	63,5	68,5	72,2	74,9	80,5	83,1	89,0	91,5	90,1	87,2	75,4	64,1	56,8	52,6	43,6	36,8
620	34,9	36,7	36,3	34,9	33,3	35,1	41,8	55,3	63,8	68,4	72,0	75,2	81,0	84,0	89,4	91,2	91,8	87,8	75,6	64,2	57,2	52,7	43,7	36,7
619	35,4	36,7	36,3	34,9	33,2	35,5	42,6	56,6	65,7	68,6	72,4	76,2	81,8	84,0	89,7	92,5	92,1	87,6	75,4	64,4	57,2	52,8	43,3	36,8
618	35,7	36,6	36,4	34,7	33,3	36,0	42,6	57,3	63,0	70,3	71,5	76,0	81,1	84,1	89,8	94,0	92,4	87,7	75,4	64,4	57,1	52,6	43,4	36,8
617	35,3	36,6	36,4	34,9	33,3	36,9	43,0	58,0	61,6	70,7	73,8	77,2	81,4	84,8	90,4	95,5	92,4	89,2	75,3	64,3	57,0	52,6	43,5	36,4
616	35,8	36,6	36,4	34,7	33,2	37,3	43,4	58,0	67,1	72,6	73,1	77,1	79,1	84,5	90,3	94,6	92,6	88,4	75,5	64,3	57,4	52,8	43,3	36,3
615	35,2	36,4	36,4	34,7	33,2	37,8	43,9	60,1	68,3	71,9	74,4	77,2	82,6	86,0	90,4	95,8	93,1	88,2	75,6	64,3	57,6	52,7	43,4	36,0
614	36,4	36,6	36,7	35,5	33,2	38,0	44,8	61,6	68,6	72,2	74,3	78,7	83,5	85,5	91,3	94,8	92,8	88,8	75,4	64,2	57,8	52,6	43,4	36,3
613	37,7	36,5	36,7	35,3	33,1	38,2	45,2	61,7	67,0	73,9	75,1	78,6	84,1	85,6	91,0	96,1	93,4	88,5	75,3	64,5	57,9	52,7	43,4	36,7
612	37,5	36,8	36,6	35,1	33,0	37,8	44,7	62,8	69,8	73,7	72,4	79,0	84,7	86,2	91,0	95,4	93,5	87,5	75,2	64,5	57,8	52,7	43,3	36,8
611	37,5	36,8	36,5	34,7	32,6	38,6	46,6	63,8	71,6	75,2	72,2	79,7	84,2	85,7	91,5	96,2	93,4	87,0	75,2	64,6	57,7	52,8	43,3	36,8
610	37,6	36,7	36,3	34,3	32,5	38,7	46,3	63,8	71,6	76,2	77,3	80,6	85,6	86,8	91,6	95,3	93,9	87,7	75,5	64,4	58,0	52,8	43,3	36,9
609	37,8	36,7	36,0	34,7	32,7	38,4	46,8	65,6	75,2	76,9	77,3	80,0	85,4	86,4	92,0	95,9	93,6	87,3	75,9	64,5	58,1	52,7	43,8	37,5
608	37,6	36,6	35,9	34,8	32,9	38,1	47,5	64,8	72,8	77,8	76,2	80,0	86,0	87,5	91,6	95,1	93,7	86,4	75,8	64,1	57,9	52,8	43,9	37,9
607	37,7	36,4	36,6	34,9	32,9	40,6	47,9	65,5	72,6	78,0	77,3	79,4	86,6	88,3	91,6	95,1	94,0	86,6	75,9	64,4	57,7	52,9	43,9	37,6
606	37,8	35,9	36,9	34,8	33,5	41,5	49,5	65,5	73,6	78,6	78,7	82,1	86,8	87,9	91,3	95,0	94,0	89,4	75,7	64,8	58,2	52,0	43,1	37,4
605	38,0	35,5	36,4	35,2	33,2	39,6	49,9	63,1	79,4	78,3	79,7	82,6	86,8	87,7	91,0	94,7	93,7	86,2	75,5	64,4	57,8	51,2	43,3	37,5
604	37,8	36,5	36,2	35,7	33,4	40,5	50,1	66,5	81,2	79,4	79,0	83,6	87,2	87,2	90,8	94,6	94,2	86,4	75,3	64,0	56,9	50,5	42,6	38,6
603	37,6	37,1	36,3	35,0	34,8	40,1	48,9	65,0	80,2	81,7	80,0	85,7	86,7	88,2	92,0	95,6	93,8	90,7	76,0	64,8	55,4	51,6	41,0	37,5
602	37,1	37,2	36,2	34,2	39,7	40,0	56,6	71,0	80,4	81,3	82,1	87,4	87,3	89,9	93,3	95,5	93,1	90,6	76,2	63,7	56,7	48,4	40,7	37,2
601	34,9	38,3	35,7	34,5	39,2	42,3	56,8	73,3	82,0	81,7	83,6	86,8	88,6	90,6	94,5	94,6	91,8	88,5	74,5	62,7	57,4	44,8	41,1	35,5
600	35,1	39,1	35,6	34,8	44,2	44,7	47,3	74,8	81,6	84,6	85,3	86,2	92,2	92,9	94,9	94,8	94,7	84,9	74,4	63,7	56,2	49,6	41,7	34,5

Figure 3.38: Results from the ScaleX 2015 experiment focusing humidity measurements. Individual entries in the matrix show the average humidity calculated from in-situ measurements performed by the flying ensemble in three different positions (cf. Figure 3.35). Measurement flights were repeated on each full hour during the course of a day, resulting in 24 runs in total. Measurement depicted start at a height of $600m$ ASL and range up to $699m$ ASL. Colors indicate the development of the humidity gradient during measurements performed withing all 24 runs (data misses because of data inconsistencies for run 7 at 20:00 o'clock at height $699m$ ASL).

that a temperature inversion must be present in the NBL, because at daytime temperatures near the ground (i.e., close to $600m$ ASL) are *higher* (deeper red) than in the skies (lighter red), and at nighttime temperatures near the ground are *lower* (deeper blue) than in the skies (lighter blue). Figure 3.37b confirms that finding when we investigate the relative temperature gradient for each run. Doing so reveals the occurrence of a temperature inversion, beginning at 19:00 local time of June 30th and ending at 07:00 local time of July 1st. We can conclude that, because in each run within this time window (i.e., during nighttime), color gradients range *from deep-blue* (low temperature) at heights near ground *to deep-red* (high temperatures) at heights in the skies. During daytime instead, we can see the typical color gradient from deep-red colors (i.e., higher temperatures) near the ground to deep-blue colors (i.e., lower temperatures) near the skies with some heat fluxes at different heights, most probably caused by thermal winds in the morning.

In Figure 3.38, we can see an effect caused by the temperature inversion. For every entry in the matrix, we depict the average measured relative humidity h_{rel}. We are interested in measuring humidity, e.g., when we want to analyze the current PM concentration in the NBL because high humidity has a direct influence on the quality of particle counters used to determine the PM concentration [Jayaratne et al., 2018] and the total amount of PM particles of one size (high humidity lets particles grow in size [Hernandez et al., 2017]). We depict measurements with a *low value of* h_{rel} *with deep red color* and measurements with a *high value of* h_{rel} *with deep blue color*. We know that the maximum possible level of humidity h_{max} transported by a specific volume of air (e.g., measured in kilogram per cubic meter kg/m^3) is limited by the respective temperature in that volume [Dyck and Peschke, 1983]. Thus, higher tempered volumes of air can take up higher amounts of water. In a closed system during the occurrence of an inversion, we would assume to see lower values of h_{rel} in higher heights than on the ground level. We cannot make such an assumption when analyzing highly dynamic meteorological systems like that is present in DE-Fen where, among many other factors, a highly pronounced vegetation (grassland) influences the humidity fluxes [Brenner et al., 2018]. However, what we can still see in Figure 3.38 is a change in the slope of humidity saturation on different heights, related to the relatively higher temperatures in higher heights when compared to that of lower heights. The measured relative humidity increases with a factor of 2.95 close to lower heights and with a factor of 2.30 at higher heights. At $611m$ ASL, e.g., measured values change from from 32.6% at 18:00 o'clock local time (minimum measured value) to 96.2% at 05:00 o'clock local time (maximum measure value) — we do not take values to close to the overgrown surface, i.e., at $600m$ ASL. The factor for $615m$ ASL thus is $2.95 = 96.2\%/32.6\%$. At $699m$ ASL, e.g., measured values change from 31.9% at 19:00 o'clock local time (minimum measured value) to 73.5% at 06:00 o'clock local time (maximum measured value). The factor for $699m$ ASL thus is $2.30 = 73.5\%/31.9\%$. We support these findings with additional plots and evaluations in Appendix A.1.

Finally, in Figure 3.39 we depict three temperature profiles derived by the whole ensemble, i.e., all three agents, during an exemplary flight. In this figure, we analyze the feasibility of detecting the temperature inversion with lightweight onboard sensors we encapsulated in the $\text{SDH}_{\text{SHT75}}$-prototype in general. Figure 3.39 shows us that every one of the three agents detected the inversion layer in height between $70-80m$ AGL. For each agent, the respective temperature gradient's development changes its continuous increase, from approximately $18.0°C - 18.5°C$ at $20m$ AGL to approximately $21.5°C$ (green) – $21.6°C$ (blue) at $70 - 80m$ AGL, back to a decrease to $21.2°C$ (red) – $21.5°C$ (blue) at $100m$ AGL. Thereby, we can see that agents detect the temperature inversion at different heights and with different expression at their respective

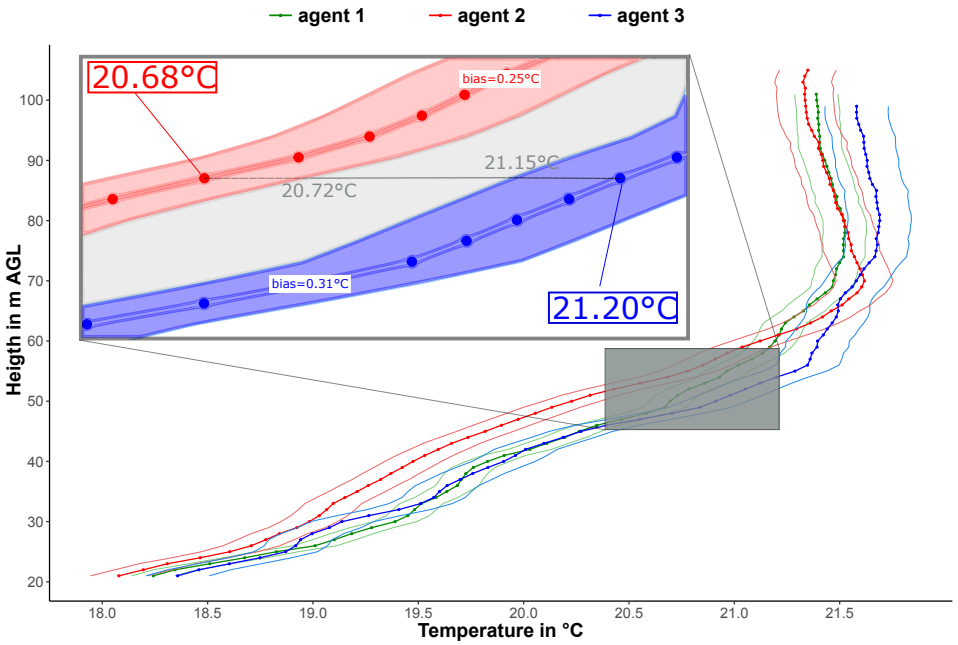


Figure 3.39: Synchronous temperature measurements achieved by the individual agents in the ensemble. Bold colored lines indicate the average measured temperature for the respective height agent-specific, lighter colored lines the agent-specific standard deviation of the derived measurements. Already small spatial variations can result in very different temperature measurements on the same heigth (here at $54m$ AGL).

measuring positions. Thus, the expression of the temperature inversion can differ already at small distances like that we covered within our experimental site of approximately $200m$. During all measurement flights, the measured temperature at the same height for the same time derived at the three different positions ranged from $0.06°C$ to $0.7°C$. Thus, to determine the spatial expansion of the temperature inversion, using flying ensembles instead of only individual UAV is urgently necessary when relying on light-weight and thus using relatively imprecise sensors for in-situ measurements is inevitable. One agent alone would not necessarily be able to detect the phenomena because changes in temperature measurements could also be caused by the inaccuracy of single sensors. When analyzing the measured temperature gradient of agent 3 (blue) in Figure 3.39, we can see only a minor change in the slope starting at approx. $80m$ AGL. While we know from the similar trend measured by agent 1 and agent 2 that this change already is due to the temperature inversion influencing temperature inversion on the named height, agent 3 alone could not decide whether the measured change in the slope is only caused by the sensor-inherent measurement inaccuracy.

3.4.1.4 Conclusion

Summed up, the ScaleX 2015 experiment showed that the use of flying ensembles for the collective and distributed measurement of environmental parameters is feasible. Based on our results, relatively precise tracking of the NBL is possible despite low-price and low-weight sensors when using ensembles for the measurements (cf. Figure 3.37b). Thus, we can use flying

ensembles to find meteorological phenomena, determine their impact on other parameters of interests (like, e.g., PM concentration or humidity in Figure 3.38), and decide on appropriate reactions according to the data derived by the flying ensemble. We further see that for handling the effects of meteorological phenomena in situations with high spatial variability like the temperature inversion, using more than one single agent is urgently necessary because of inevitable senor inaccuracies (cf. Figure 3.39).

3.4.2 Field Experiment ScaleX 2016: Automating Plan Execution in Real-World SCORe Missions in the Context of Environmental Monitoring

In the second experiment during ScaleX 2016, we demonstrated the feasibility of working through plans from a SCORe mission with real robots implementing the concepts from our Multipotent System reference architecture in the field. We thereby aimed at reducing the overhead a user of a Multipotent System has when instructing an ensemble for operating in that plan compared to our ScaleX 2015 experiment. Therefore, we realized more elements from our reference architecture for Multipotent Systems to automate the individual agent control as well as the ensemble coordination in our prototypes. Caused by the experiment design and the required measuring hardware (i.e., one concrete SDH we used was significantly heavier than the maximum payload of our UAVs), we integrated a non-flying robot besides our flying agents in the Multipotent System. Thereby, we were able to show the extensibility of the design of the reference architecture for Multipotent Systems to non-exclusively flying ensembles, i.e., such consisting of a mixture of robot hardware like UAV, UGV, etc. Again, the experiment was carried out at the DE-Fen, this time on July 15th of 2016. We provide a video of the experiment on GitHub and YouTube[3] (video *Flying robot ensemble in action at the ScaleX 2016 geographic measurement campaign*).

3.4.2.1 Experiment Design

In the partial SCORe mission we designed for the ScaleX 2016 field experiment, we aimed at generating large-scale temperature profiles in the Atmospheric Boundary Layer (ABL) at flexible and even airborne locations. The retrieved large-scale temperature profiles were meant to be used for the validation of measurements performed by other sensor systems located at DE-Fen, e.g., the wireless SoilNet underground humidity sensor network [Wolf et al., 2017; KIT IMK/IFU, 2018]. For generating the temperature profiles, we used the Distant Temperature Sensing (DTS) technology. With the DTS technology, temperature measurements can be made within every meter of a fiber-optic cable whose length can flexibly be extended for large measuring distances [Sensornet, 2018a]. Usually, DTS measuring devices are installed on the ground in a fixed way [Zeeman et al., 2014]. In contrast to that, our goal in the ScaleX 2016 field experiment was to use an ensemble implementing the Multipotent Systems reference architecture to show its potential for making the DTS technology flexibly usable. We achieved this by realizing the coordinated and cooperative transport of the DTS sensing unit, i.e., the fiber-optic cable. We depict a rendered sketch of the idea in Figure 3.40.

We designed a respective partial SCORe mission involving multiple agents from our Multipotent System to realize the idea. Due to the huge technical overhead, we included only one

[3] `https://github.com/isse-augsburg/ensemble-programming` or `https://github.com/kosakoliver/ensemble-programming` or `https://www.youtube.com/user/ISSELabs`

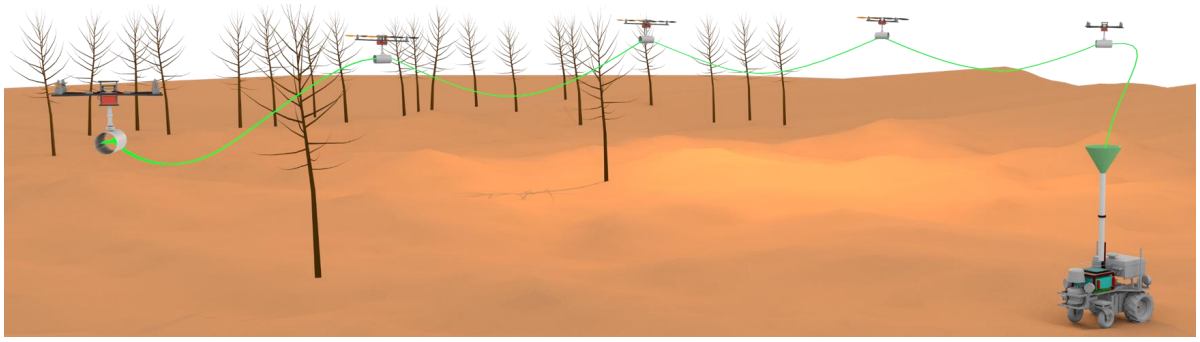
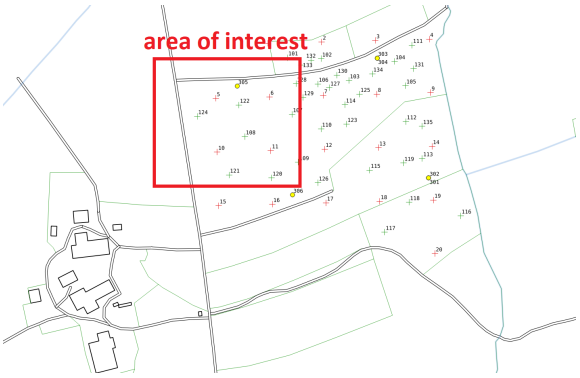


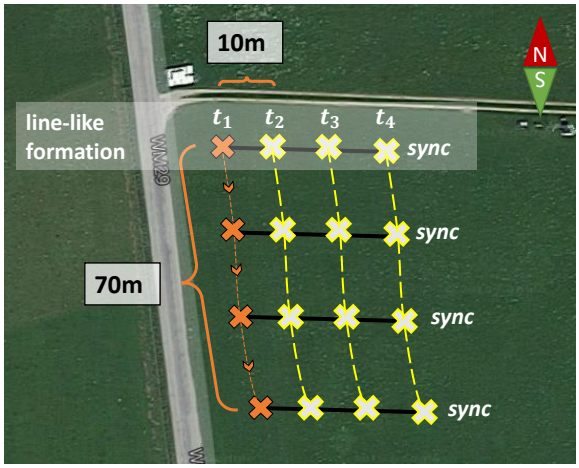
Figure 3.40: Rendered sketch of the ScaleX 2016 experiment. The ensemble consists of six agents (five flying agents equipped with SDH encapsulating UAV and one driving agent equipped with an SDH encapsulating an UGV). The ensemble collectively carries the DTS measuring device (a fiber-optic cable attached to an evaluation unit) we can use for large-scale and airborne temperature measurements. While flying agents can carry the glass fiber-optic cable belonging to the DTS measuring device (i.e., the sensing unit) into airborne heights, the driving agent can carry the heavy evaluation unit to which the fiber-optic cable is attached.

plan $\rho_{\text{SCALEX2016}}$ in this SCORe mission. In $\rho_{\text{SCALEX2016}}$, we required an ensemble to carry the fiber-optic cable (i.e., the sensor of the DTS measuring device) cooperatively along a defined route in DE-Fen (cf. spatial conditions depicted in Figure 3.41). We defined that route to lead over a part of the SoilNet system (cf. Figure 3.41a) to allow for the cross-validation of data retrieved by the two sensor systems. From a top-down perspective (i.e., flattened to a 2-dimensional view), we required the ensemble to build a line-like formation first. We achieved this by designing the plan including a respective amount of tasks for an ensemble encoding the relevant positions in this line-like formation. While the number of tasks on $\rho_{\text{SCALEX2016}}$ was flexible in principle, for most of our experimental runs, we used four tasks $t_{1,\dots,4}$ only as depicted in Figure 3.41b. This helped us drastically reduce the maintenance overhead caused by the necessary hardware because scaling the number of tasks in $\rho_{\text{SCALEX2016}}$ also requires an ensemble $\mathcal{E}^{\text{SCALEX2016}}$ involving more agents to work on the plan. Second, we required $\mathcal{E}^{\text{SCALEX2016}}$ to move along the defined route from north to south for a distance of $70m$ while synchronizing their positions frequently in between (cf. sync markers in Figure 3.41b). Synchronization was necessary because during working on the plan, agents in the ensemble were physically connected by the fiber-optic cable they cooperatively carried (cf. Figure 3.41c), allowing for only a little discrepancy in execution speed. While we designed the plan to move agents with a distance of $10m$ in the line-like formation, we added some additional cable between them (cf. the fiber-optic cable depicted in red in Figure 3.41c). Nevertheless, if one agent in $\mathcal{E}^{\text{SCALEX2016}}$ had moved much faster than another, the fragile fiber-optic sensor would have been destroyed, leading the experiment to fail without any measurements.

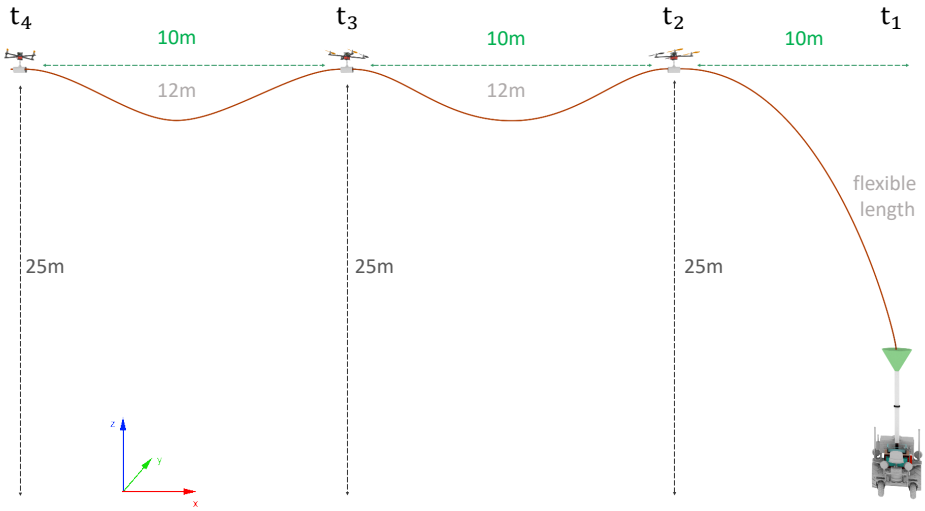
To achieve not only mobile but also airborne DTS measurements, we further defined the movement heights for all paths in tasks $t_{1,\dots,n}$ within $\rho_{\text{SCALEX2016}}$ to be at $25m$. We required each agent to allocate any of the tasks to provide the capability $c^{p}_{\text{MV-POS}}$ for moving to a given position enabling it to move along the positions of the respective path defined in the task. Apart from $c^{p}_{\text{MV-POS}}$, the set of required capabilities for the tasks in the plan differed between task t_1 and tasks $t_{2,\dots,n}$ due to the specific DTS measuring technology: While for tasks $t_{2,\dots,n}$, we required the respective agent allocating it to provide the capability for carrying the fiber-



(a) SoilNet sensor placement, picture from KIT IMK/IFU [2018]



(b) movement pattern (top down perspective), map data © 2021 GeoBasis-DE/BKG (© 2009)



(c) movement pattern (rear perspective),

Figure 3.41: Spatial conditions for the SCORe mission during our ScaleX 2016 field experiment at DE-Fen, performed on July 15th of 2016. Figure 3.41a shows the distribution of the underground sensors belonging to the SoilNet measuring system [Fersch et al., 2020]. Figure 3.41b shows the movement path in a plan consisting of four tasks $t_{1,\ldots,4}$ leading from north to south over a part of the SoilNet system. Distances between the positions in $t_{1,\ldots,4}$ were $10m$, the total length of each path was $70m$ (with synchronization points *sync* in between). Differently colored tasks require different sets of capabilities. Tasks t_1 and $t_{2,3,4}$ differed in their capabilitiy requirements (cf. Table 3.13).

Table 3.13: The task requirements of tasks t_1, ..., t_n in the plan $\rho_{\text{SCALEX2016}}$ during the partial SCORe mission designed for the ScaleX 2016 field experiment.

task t	required capabilities $\mathcal{C}_t$
t_1	$\{c^p_{\text{M-DTS}}, c^p_{\text{MV-POS}}, c^p_{\text{M-POS}}\}$
$t_{2,...,n}$	$\{c^p_{\text{CARRY-DTS}}, c^p_{\text{MV-POS}}, c^p_{\text{M-POS}}, c^p_{\text{M-TEMP}}\}$

optic cable $c^p_{\text{CARRY-DTS}}$ and create comparative temperature measurements in-situ with $c^p_{\text{M-TEMP}}$, in task t_1, we required the respectively agent allocating it to provide the capability $c^p_{\text{M-DTS}}$ for executing measurements instead (cf. Table 3.13).

Figure 3.41c illustrates a possible setup of an ensemble $\mathcal{E}^{\text{SCALEX2016}}$ capable of executing the so defined plan $\rho_{\text{SCALEX2016}}$. Due to hardware weight limitations, we further describe in the next paragraph, we required one agent in $\mathcal{MS}_{\text{SCALEX2016}}$ to be configured with an SDH encapsulating an UGV instead of an UAV for enabling $\mathcal{MS}_{\text{SCALEX2016}}$ to generate a valid allocation for task t_1. This hardware restriction required us to introduce a minimal-invasive abstraction on the SEMANTIC HARDWARE LAYER, masking the difference in the used hardware to higher layers in our Multipotent System reference architecture. Doing so allowed us to maintain our goal to program flying ensembles but also required us to make appropriate further adjustments to the experiment setup, which we depict in Figure 3.41c. We required the fiber-optic cable length between the agent allocating task t_1 and allocating task t_2 to be of adjustable length. Second, we required the agent configured with the SDH encapsulating the UGV to interpret positions defined within the path in the task it allocates like they were located on the surface level (i.e., neglect the commanded altitude encoded in the positions of t_1). All other tasks $t_{2,...,n}$ could be allocated by other agents configured with SDH encapsulating an UAV. Making this abstraction allowed us to evaluate the interplay of our Multipotent System reference architecture concepts while also widening our focus to driving robots. In our prototypical implementation for the ScaleX 2016 field experiment, we thus could give proof of the correct conceptual integration required for the autonomous Ensemble Program generation from a plan, the self-aware task allocation, and the coordinated execution of that plan.

3.4.2.2 Software and Hardware Prototypes

Our ScaleX 2016 field experiment used a prototypical instance of a Multipotent System, consisting of up to five agents configured with different sets of SDH. We used five different SDH ($\text{SDH}_{\text{AQ-R}}$, $\text{SDH}_{\text{INNOK}}$, $\text{SDH}_{\text{DTS-C}}$, $\text{SDH}_{\text{DTS-M}}$, $\text{SDH}_{\text{TEMOD}}$, cf. Table 3.14) providing different levels of self-descriptive information to the agents.

- The $\text{SDH}_{\text{AQ-R}}$-prototype (cf. Figure 3.42a) encapsulated an UAV based on frames from *rOsewhite* [Pietzsch, 2018] integrated with an Autoquad flight controller [Autoquad, 2018]. It provided the relevant capabilities $c^p_{\text{MV-POS}}$ and $c^p_{\text{M-POS}}$ in a hard-coded but self-descriptive manner.
- The $\text{SDH}_{\text{INNOK}}$-prototype (cf. Figure 3.42b) encapsulated an UGV based on an Innok Heros [Robotics, 2018] rover. It provided the relevant capabilities $c^p_{\text{MV-POS}}$ and $c^p_{\text{M-POS}}$ in a hard-coded but self-descriptive manner.
- The $\text{SDH}_{\text{DTS-C}}$-prototype (cf. Figure 3.43b) encapsulated a gripper for the fiber-optic cable built from specialized flexible modules to avoid possible damage done to the fiber-optic

(a) exemplary flying agent, configured for $t_2, ...t_n$ (b) driving agent, configured for t_1

Figure 3.42: Exemplary flying and driving agents used in the Scalex 2016 field experiment. Each flying agent was equipped with one $\text{SDH}_{\text{AQ-R}}$-prototype, one of two possible $\text{SDH}_{\text{TEMOD}}$-prototypes, and an $\text{SDH}_{\text{DTS-C}}$-prototype, allowing it to allocate tasks $t_{2,...,n}$. The driving agent was equipped with one $\text{SDH}_{\text{INNOK}}$-prototype (providing the capabilities $c^p_{\text{MV-POS}}$ and $c^p_{\text{M-POS}}$) and an $\text{SDH}_{\text{DTS-M}}$-prototype (providing the capability $c^p_{\text{M-DTS}}$), allowing it to allocate tasks t_1. All pictures provided by © Seubert.

(a) $\text{SDH}_{\text{DTS-M}}$-prototype, picture from [Sensornet, 2018b] (b) $\text{SDH}_{\text{DTS-C}}$-prototype, picture provided by © ISSE (c) $\text{SDH}_{\text{TEMOD}}$-prototype

Figure 3.43: SDH-prototypes used during ScaleX 2016. Figure 3.43a shows the Oryx+ DTS measurement device [Sensornet, 2018b] we used for our $\text{SDH}_{\text{DTS-M}}$-prototype. Figure 3.43b shows the $\text{SDH}_{\text{DTS-C}}$-prototype encapsulating a flexible gripping module for carrying the DTS sensor (i.e., the fiber-optic cable). Figure 3.43c shows the $\text{SDH}_{\text{TEMOD}}$-prototype we designed especially for environmental measurements. We mounted two Temod boards with temperature sensors on the STM-board and placed them in the airflow pathway. Thus, we can place the $\text{SDH}_{\text{TEMOD}}$-prototype underneath the rotors of an UAV to use the down-wash for optimal ventilation.

Table 3.14: Overview on the set of $\mathcal{SDH}_{\text{SCALEX2016}}$ including their capabilities we used during the ScaleX 2016 field experiment.

SDH $\in$ $\mathcal{SDH}_{\text{SCALEX2016}}$	description	capabilities	WEIGHT
$\text{SDH}_{\text{AQ-R}}$	an UAV	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$	$-1.5gk$
$\text{SDH}_{\text{INNOK}}$	an UGV	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$	$-200.0kg$
$\text{SDH}_{\text{DTS-C}}$	fiber-optic cable gripper	$\{c^p_{\text{CARRY-DTS}}\}$	$+0.2kg$
$\text{SDH}_{\text{DTS-M}}$	DTS measuring unit	$\{c^p_{\text{M-DTS}}\}$	$+8kg$
$\text{SDH}_{\text{TEMOD}}$	a temperature sensor	$\{c^p_{\text{M-TEMP}}\}$	$0.05kg$

cable during the movement of the agent carrying it. It provided the relevant capability $c^p_{\text{CARRY-DTS}}$ without any self-description (required implicit information at the respective agent).

- The $\text{SDH}_{\text{DTS-M}}$-prototype (cf. Figure 3.43a) encapsulated the DTS evaluation unit, an Oryx DTS measurement device [Sensornet, 2018b]. It provided the relevant capability $c^p_{\text{M-DTS}}$ without any self-description (required implicit information at the respective agent).
- The $\text{SDH}_{\text{TEMOD}}$-prototype(cf. Figure 3.43c) encapsulated a Temod temperature sensor coupled with a PT-1000 sensing module [GROSENS INSTRUMENTS GmbH, 2018]. It provided the capability $c^p_{\text{M-TEMP}}$ in a self-descriptive manner.

The set of SDH we configured each agent with enabled the respective agent to execute the capabilities required in the tasks defined in the partial SCORe mission. We pre-configured four of the five agents with one $\text{SDH}_{\text{AQ-R}}$, one $\text{SDH}_{\text{DTS-C}}$, and two $\text{SDH}_{\text{TEMOD}}$ each (cf. Figure 3.42a, i.e., those agents became flying agents. The fifth agent, we configured with one $\text{SDH}_{\text{INNOK}}$ and one $\text{SDH}_{\text{DTS-M}}$ (cf. Figure 3.42b), i.e., that agent became a driving agent. That way, our Multipotent System consisting of five agents handled 18 SDH in total during the ScaleX 2016 field experiment (cf. Table 3.15).

To host the run-time environment for our prototypes, including the software for agents in the Multipotent System, we used an Odroid XU4 [Hardkernel, 2018], running a Linux-based Operating System. Each agent implemented the concepts from the ENSEMBLE LAYER, the AGENT LAYER, and the SEMANTIC HARDWARE LAYER from our Multipotent System reference architecture. In the implemented prototypical state, all agents were able to coordinate and participate in Ensemble Formation autonomously, task allocation, plan execution, capability coordination, and capability execution. The set of $\mathcal{SDH}_{\text{SCALEX2016}}$ we had available for configuring the Multipotent System during the experiment in ScaleX 2016 included only one $\text{SDH}_{\text{DTS-M}}$-prototypes providing the capability $c^p_{\text{M-DTS}}$ with a total weight of $+8kg$. The SDH encapsulating UAV we had available during our ScaleX 2016 field experiment (i.e., our $\text{SDH}_{\text{AQ-R}}$-prototypes) could not carry the $\text{SDH}_{\text{DTS-M}}$-prototype unless we used more powerful devices compared to our ScaleX 2015 experiment. Thus, we were required to soften our scope of using only flying ensembles and introduced additional robot hardware to the Multipotent System providing the required weight carrying capacities. We did this in the form of the $\text{SDH}_{\text{INNOK}}$ encapsulating a powerful UGV (cf. Figure 3.42b). We integrated the $\text{SDH}_{\text{INNOK}}$-prototype to act like an SDH encpsulating an UAV, i.e., providing the typical capabilities $c^p_{\text{MV-POS}}$, $c^p_{\text{MV-VEL}}$, $c^p_{\text{M-POS}}$, and $c^p_{\text{M-VEL}}$, but with restrictions to $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-VEL}}$. Obviously an UGV cannot move to positions commanded

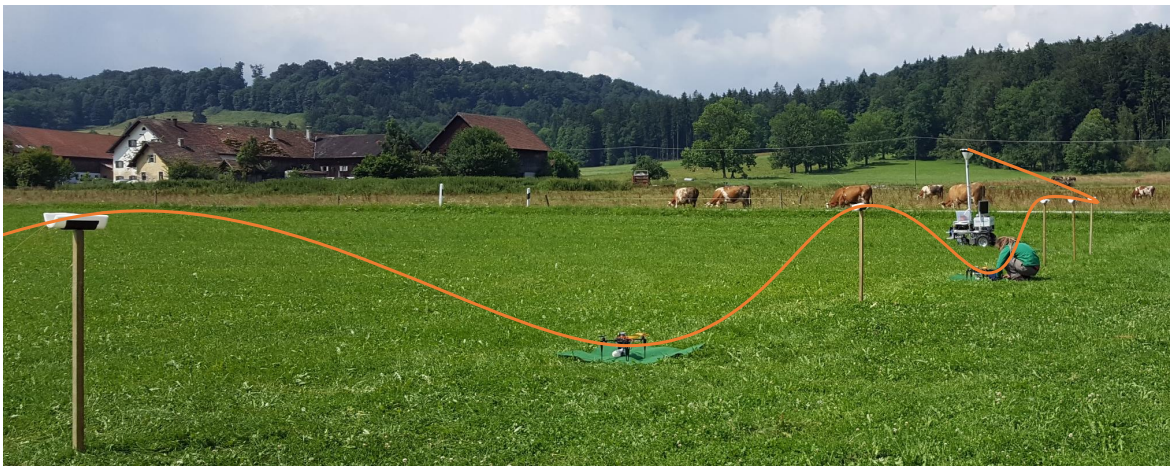
Table 3.15: Configuration of agents concerning their set of connected SDH-prototypes and the resulting provided capabilities during the ScaleX 2016 field experiment.

agent $\alpha \in \mathcal{A}_{\text{SCALEX2016}}$	the agent's SDH $\mathcal{SDH}_\alpha$	the agent's provided capabilities $\mathcal{C}_\alpha$
α_1	$\{\text{SDH}_{\text{INNOK}}, \text{SDH}_{\text{DTS-M}}\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$
$\alpha_{2,\dots,4}$	$\{\text{SDH}_{\text{AQ-R}}, \text{SDH}_{\text{DTS-C}}, \text{SDH}_{\text{TEMOD}}\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{CARRY-DTS}}, c^p_{\text{M-TEMP}}\}$
(optional $\alpha_{5,\dots,n}$)	$\{\text{SDH}_{\text{AQ-R}}, \text{SDH}_{\text{DTS-C}}, \text{SDH}_{\text{TEMOD}}\}$	$\{c^p_{\text{MV-POS}}, c^p_{\text{MV-VEL}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}, c^p_{\text{CARRY-DTS}}, c^p_{\text{M-TEMP}}\}$

in the parameters for $c^p_{\text{MV-POS}}$ with a value of the z-component having $z > 0$ or execute $c^p_{\text{MV-VEL}}$ with a parameter having the z-component set to a value $z \neq 0$. To overcome this problem for our ScaleX 2016 experiment, we made an abstraction for the implementation of $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-VEL}}$ for the $\text{SDH}_{\text{INNOK}}$ directly on SEMANTIC HARDWARE LAYER to interpret all parameters of $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-POS}}$ as if their z-component was set to $z = 0$. The agent configured with $\text{SDH}_{\text{INNOK}}$ had no knowledge of this abstraction and thus was able to coordinate the execution of $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-VEL}}$ without restriction. Thus, in the plan for the ScaleX 2016 field experiment, we did not differentiate in the possible range the capability $c^p_{\text{MV-POS}}$ could be executed by agents, i.e., we did not differentiate in *driving* and *flying* concerning the $c^p_{\text{MV-POS}}$ capability. Thereby, also, the agent configured with $\text{SDH}_{\text{INNOK}}$ could allocate a task from the plan. More precisely, it could exclusively allocate the task t_1 (cf. Figure 3.41c) because it could not provide $c^p_{\text{CARRY-DTS}}$ all other tasks $t_{2,\dots,4}$ required. Because there were no other agents configured for providing $c^p_{\text{M-DTS}}$ and we did not take into account the possibility of physical reconfiguration during our ScaleX 2016 field experiment, the only agent that could allocate t_1 thus was the agent having the $\text{SDH}_{\text{INNOK}}$-prototype in its set $\mathcal{SDH}_\alpha$.

3.4.2.3 Experiment Execution and Results

We pre-configured our agents according to the entries in Table 3.15. During the experiment setup, we were required to ensure that the fiber-optic cable would not get stuck in the vegetation present in DE-Fen on the one hand, and it would not be stretched too much by the ensemble $\mathcal{E}^{\text{SCALEX2016}}$ throughout working through the whole plan $\rho_{\text{SCALEX2016}}$. Therefore, we carefully placed each agent individually before each introduction of $\rho_{\text{SCALEX2016}}$ (cf. Figure 3.44). Further, we also pre-connected the fiber-optic cable to the grippers within the $\text{SDH}_{\text{DTS-C}}$-prototypes for agents $\alpha_{2,3,4}$ (and $\alpha_{5,\dots,n}$ in the respective extended experiments). We required to restrict the system in its flexibility at this point as gripping the very small fiber-optic cable otherwise would have increased the technical hurdles disproportionate to the actual goal we wanted to achieve. Moreover, we introduced an optimization criterion in the task allocation mechanism specifically designed for the ScaleX 2016 experiment: Because each agent $\alpha_{2,\dots,n}$ could allocate each of the tasks $t_{2,\dots,n}$, unfortunate allocations could do damage to the fiber-optic cable (cf. Figure 3.45). For avoiding that, we requested the agents to additionally provide their positions when sending proposals for tasks $t_{1,\dots,n}$ in the $\rho_{\text{SCALEX2016}}$, allowing to ensure an allocation of tasks to the individually best-positioned agents (cf. Figure 3.45a). That way, and by position-

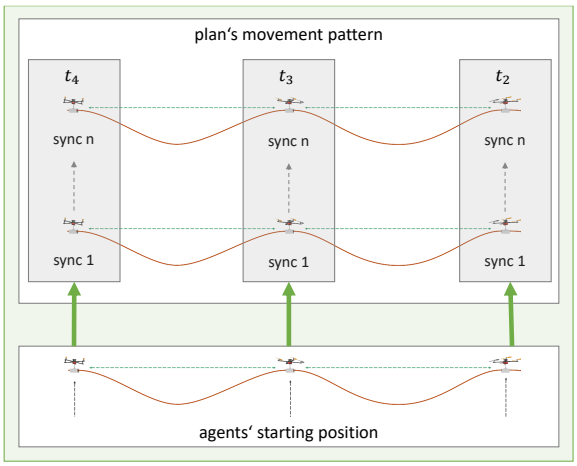


(a) ensemble preparation for the subsequent plan execution during the ScaleX 2016 field experiment, ©ISSE

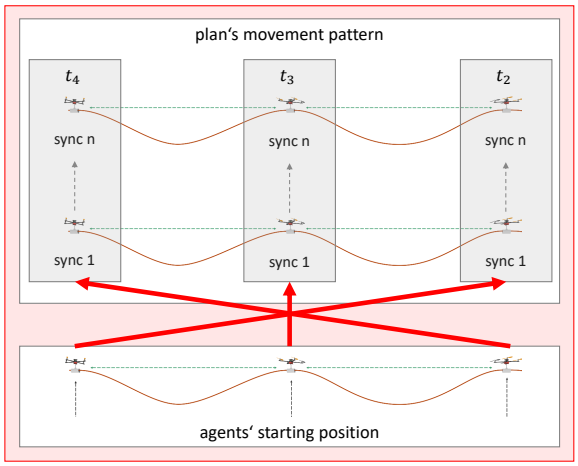
(b) flying agents (α_2, ..., α_4) setup, © Seubert

(c) driving agent (α_1) setup, © Seubert

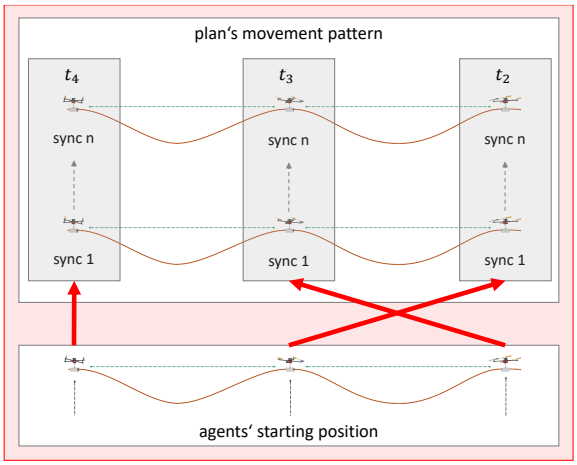
Figure 3.44: Setting up the hardware before instructing the Multipotent System with a plan in the ScaleX 2016 field experiment aiming at airborne, mobile DTS-measurements performed over the SoilNet system (cf. Figure 3.41a). Because of the difficulties that would arise for the gripper encapsulated in $\text{SDH}_{\text{DTS-C}}$ while picking up the fiber-optic cable, we pre-connected the respective SDH of agents $\alpha_{2,3,4}$ to the fiber-optic cable manually before introducing the plan. In Figure 3.44a, we highlighted the fiber-optic cable with an orange-colored line.



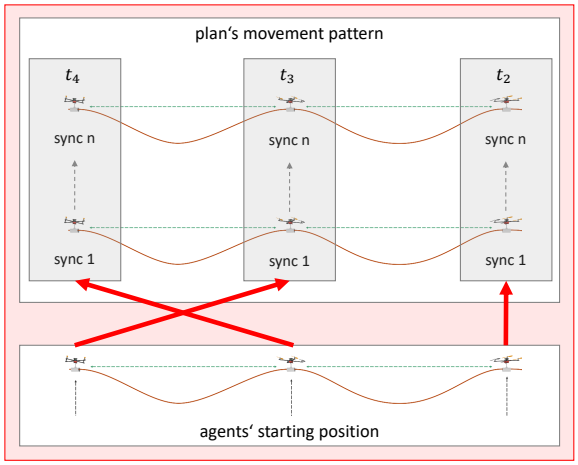
(a) allocation 1, avoiding damage



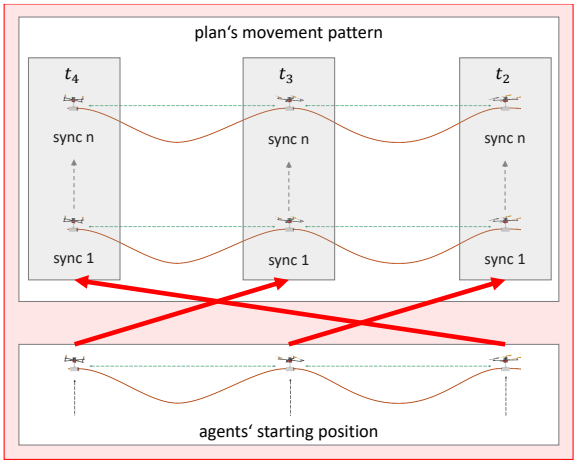
(b) allocation 2, harming the fiber-optic cable



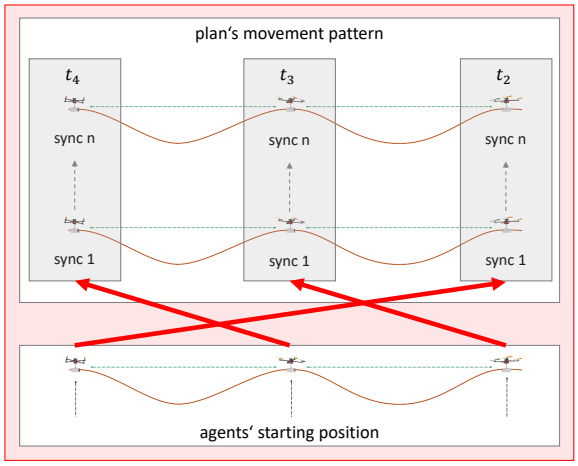
(c) allocation 3, harming the fiber-optic cable



(d) allocation 4, harming the fiber-optic cable



(e) allocation 5, harming the fiber-optic cable



(f) allocation 6, harming the fiber-optic cable

Figure 3.45: All possible six (faculty of 3, i.e., 3!) different allocations of tasks $t_{2,3,4}$ to agents providing the required capabilities (i.e., $\alpha_{2,3,4}$). The figures visualizes the different effect allocations can have on the fragile fiber-optic cable, assuming that grippers encapsulated in $\text{SDH}_{\text{DTS-C}}$ that agents are configured with are pre-connected to the fiber-optic cable. Only the allocation in Figure 3.45a avoids harming the fiber-optic cable.

ing the agents very close to the initial positions of the respective task, we could ensure to avoid damage done to the fiber-optic cable during the preparation process of the Multipotent System before collectively working through the $\rho_{\text{SCALEX2016}}$ with an ensemble $\mathcal{E}^{\text{SCALEX2016}}$. Nevertheless and despite theses restrictions, the SO-Mechanism we used for Ensemble Formation was working as described in our reference architecture so that we could validate its functionality combined with the subsequent autonomous Program Execution.

During the execution of the plan, the additional fiber-optic cable of $2m$ length between $\alpha_{2,3,4}$ and the flexible length between α_1 and α_2 (ensured by a cable reel with automatic retraction) combined with the frequent synchronization of the executing ensemble ensured to not harm the fiber-optic cable (cf. Figure 3.46). After successfully working through the plan automatically, the formed ensemble $\mathcal{E}^{\text{SCALEX2016}}$ for the plan $\rho_{\text{SCALEX2016}}$ was dissolved automatically according to the concepts from the ENSEMBLE LAYER implemented in the ScaleX 2016 prototype of our Multipotent System reference architecture. We did not yet include all concepts from the PLAN LAYER in this prototype due to the physical complications caused by the complexity of picking up and carrying the fiber-optic cable, among others. Thus, after dissolving the ensemble automatically, we manually returned all agents allocating tasks in $\rho_{\text{SCALEX2016}}$ back to the user's position.

In Figure 3.47, we depict the airborne measurements performed by the ensemble during the execution of one exemplary plan involving four agents performed on July 15th of 2016 between $13:23:16$ o'clock and $13:27:34$ o'clock (both in UTC+2). Thus, the experiment was performed shortly after midday, where temperature gradients in the lower-ABL are warmer at the ground and cooler at higher heights. Measurements here illustrate the results achieved by agent α_1 executing its capability $c^{p}_{\text{M-DTS}}$. Agents $\alpha_{2,\ldots,4}$ did not perform DTS-based measurements but instead carried the fiber-optic cable by executing $c^{p}_{\text{CARRY-DTS}}$ instead. Figure 3.47 depicts the course of changes in temperature measurements performed by α_1 at all positions along the line-like formation built by the ensemble as a whole (i.e., from agent α_1 over α_2 and α_3 to α_4) with a distance of $1m$ in-between measurements. With the Oryx+ DTS instrument encapsulated in the $\text{SDH}_{\text{DTS-M}}$-prototype, measurements can be performed with a frequency of 0.1 Hz. Thus, we could retrieve the very fine-grained mesh of measurement positions and interpolate values in between with an appropriate movement speed. The 3-dimensional plot in Figure 3.47 shows each measurement's position along the predefined path of the ensemble (cf. Figure 3.41b), reduced on the x- and y-component. We do not need to show the z-component of the measurements' positions because we know the measurement was performed at the measurement height we designed within $\rho_{\text{SCALEX2016}}$, i.e., the measurement height stayed the same for the whole execution. Instead, we plot the respective measured temperature on the z-axis in Figure 3.47, depicting the lowest measured values in deep blue, highest measurements in deep red, and other temperatures according to the color gradient in between. This way, the time of aerial measurements between takeoff and landing can be identified in Figure 3.47. The temperature measurements of approximately 13.0°C and lower in the time during $13:23:16$ o'clock (UTC+2) and $13:27:34$ o'clock (UTC+2) compared to temperatures above 17°C on the ground level (before takeoff, after landing) indicate when $\mathcal{E}^{\text{SCALEX2016}}$ was flying. We identified the 'temperature spikes' in measurements before takeoff to be the starting positions of agents $\alpha_{2,3,4}$. The fiber-optic cable could warm up at these positions while laying on the ground directly exposed to the sun (positions in between were ventilated by the wind while hanging from the wooden piles). We can see a similar heat up after landing, where the fiber-optic cable rapidly warmed up before we could detach it from the measuring unit. In Appendix A.1, we include additional evaluations concerning the experiment also investigating the idea of making

(a) $\mathcal{E}^{\text{SCALEX2016}}$ during working on a $\rho_{\text{SCALEX2016}}$ during the ScaleX 2016 field experiment (shortly after the start)

(b) $\mathcal{E}^{\text{SCALEX2016}}$ after first *sync*, flying agents only

(c) focus on one flying agent (α_4) executing $c^{p}_{\text{CARRY-DTS}}$

Figure 3.46: Real hardware-based ensemble $\mathcal{E}^{\text{SCALEX2016}}$ executing the plan $\rho_{\text{SCALEX2016}}$ of the partial SCORe mission we designed the ScaleX 2016 field experiment. The $\mathcal{E}^{\text{SCALEX2016}}$ collectively carries a Distant Temperature Sensing (DTS) measuring unit, consisting of a fiber-optic cable and an evaluation unit. All pictures provided by © Seubert.

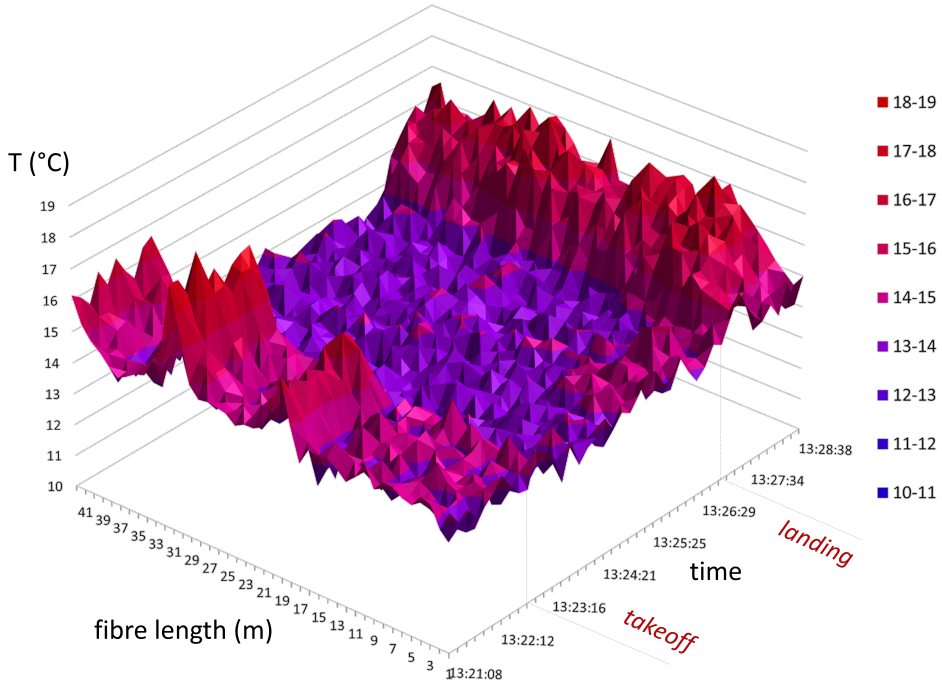


Figure 3.47: Temperature measurements derived by agent α_1 before the execution of the plan $\rho_{\text{SCALEX2016}}$ (before takeoff), during working through $\rho_{\text{SCALEX2016}}$ while allocating the task t_1 from $\rho_{\text{SCALEX2016}}$ in $\mathcal{E}^{\text{SCALEX2016}}$ (after takeoff), and after finishing $\rho_{\text{SCALEX2016}}$ (after landing) in the time between 13 : 21 : 08 and 13 : 28 : 38 o'clock on July, 15th of 2016. Different temperatures are depicted by different colors (red indicates warm, blue cold) and different values of z. Values of x describe the position along the fiber-optic cable a measurement was made at. Values of y show the development of measurements achieved during the flight (given in time UTC+2).

airborne DTS-measurements more precisise.

For the verification of the airborne measurements performed by the ensemble $\mathcal{E}^{\text{SCALEX2016}}$ collectively carrying the DTS measuring device, we used multiple $\text{SDH}_{\text{TEMOD}}$-prototypes we equipped agents $\alpha_{2,3,4}$ with. We depict the measurements performed with that $\text{SDH}_{\text{TEMOD}}$-prototypes derived during one measuring flight in Figure 3.48. Like in Figure 3.47, we can see the cooling of temperatures shortly after starting the plan (i.e., after takeoff). Also, measured temperatures coincide with the measurements we derived with the DTS system. During the plan execution, measured temperatures range between 14°C and 15°C. Before takeoff (i.e., before the plan execution), we can see the heat-up effect of the fiber-optic cable while still laying on the ground (starting positions on the ground were not ventilated). After finishing the plan and manually returning the agents to the ground, we can see the steady heating-up of measured temperatures up to 16°.

3.4.2.4 Conclusion

Our successful execution of the ScaleX 2016 experiment showed the feasibility of deploying our Multipotent System reference architecture with an adapted prototypical implementation into a

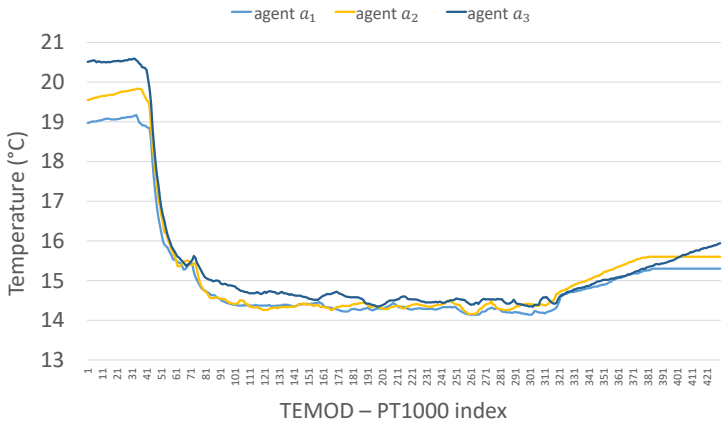


Figure 3.48: Temperature measurements performed by three agents executing $c^{p}_{\text{M-TEMP}}$ and $c^{p}_{\text{MV-POS}}$ during a SCORe mission defined for the collective transport of a fiber-optic cable. Each agent was configured with two $\text{SDH}_{\text{TEMOD}}$-prototypes for generating temperature measurements directly on-board. We depict the measurements derived by three agents $\alpha_{2,3,4}$ during an exemplary plan execution in our ScaleX 2016 field experiment. Measurements show the typical temperature gradient for the time and location measurements were performed (DE-Fen on July, 15th of 2016, early afternoon), i.e., warmer at ground level and cooler at greater heights.

real-world scenario. We demonstrated how multiple agents equipped with different prototypical SDH encapsulating versatile S&A could be easily made available to a user requiring to use their capabilities. When used appropriately, i.e., by making the necessary abstractions (here due to the physical complexity of carrying the fragile fiber-optic cable), Multipotent Systems further can provide a valuable tool for realizing innovative measuring methods. We demonstrated this by showing the feasibility of performing large-scale temperature measurements achieved by cooperative transport, i.e., by combining an established measuring system with the benefits Multipotent Systems can deliver by coordinating cooperative agents. Summed up, we proved our concept for controlling MRS with our layered software architecture used for Ensemble Formation, ensemble coordination, agents' self-awareness, and cooperative execution of plans for Multipotent System. We further demonstrated that, in general, SCORe missions can be autonomously performed by those Multipotent System.

3.4.3 Laboratory Experiment: Practicability of Physical Reconfiguration with Self-Descriptive Hardware

We base our concepts of the Multipotent System reference architecture on the availability of Self-Descriptive Hardware (SDH) and the feasibility of adapting the configuration of agents with that SDH at run-time. In this part of the evaluation concerning our Multipotent System reference architecture, we focus on the feasibility of constructing and using such SDH. We developed appropriate modular hardware devices to demonstrate the feasibility of agent reconfiguration with SDH and the so-gained flexibility. We equipped those SDH-prototypes with the required self-descriptive mechanisms necessary for empowering agents to interpret the influence of changes in their set $\mathcal{SDH}_{\alpha}$ to their set of available capabilities $\mathcal{C}_{\alpha}$. For realizing our experiment, we attached a single-board computer to different S&A, i.e., different sensor types. We encapsulated those combined modules within a 3D-printed case each. For easing

the process of exchanging that SDH, we further designed each case with magnetic connectors and a plug connection. The results from this section originate from the associated studies performed by Wanninger et al. [2018], and Eymüller et al. [2018] referring to the same reference architecture. We briefly summarize the respective results here to give proof of the feasibility for the complete set of concepts we introduced in Section 3.2.

3.4.3.1 Experiment Design

In our experiment, we wanted to show the direct influence of new agent configurations on the behavior of an agent α within a very specifically designed task that could occur in a plan within a SCORe mission. In that task, we required α to move to a given position while reacting to obstacles on the way. If an obstacle was detected, α should increase its height until it reaches a safe position, i.e., it can avoid collisions with the respective obstacle. We designed a blueprint encapsulating this behavior as a virtual capability for the sensor-based flight $c^v_{\text{MV-SENSOR}}$ provided to the agent α executing the task (cf. Section 3.2.8). To be available to α, the virtual capability $c^v_{\text{MV-SENSOR}} := \{c^p_{\text{MV-POS}} \bigstar c\}$ requires the physical capability $c^p_{\text{MV-POS}}$ as well as another measuring capability c to be available, e.g., $c = c^p_{\text{M-DIST-G}}$ for measuring the agent's distance to the ground level. For executing that task, we used one agent configured with an SDH encapsulating an UAV for providing the required capability $c^p_{\text{MV-POS}}$. We modified this agent's configuration concerning its set of other SDH by adding and removing multiple SDH-prototypes encapsulating different types of sensors to $\mathcal{SDH}_\alpha$, each providing some measuring capability c. This way, we could demonstrate the general similarity in the agent's behavior when executing the same virtual capability with different SDH encapsulating different S&A. By repeatedly executing the task with different configurations of the agent's $\mathcal{SDH}_\alpha$, we further aimed at visualizing the possible differences in the quality of execution when using SDH-prototypes encapsulating S&A of different quality. Furthermore, with our experiment, we can show the feasibility of run-time reconfigurations concerning the SDH within an agent's $\mathcal{SDH}_\alpha$ and demonstrate the actual effect of such reconfigurations. We, therefore, executed the experiment with an SDH-prototype encapsulating an ultrasonic distance sensor and a laser distance sensor. Concerning the quality of distance measurements and compared to ultrasonic distance sensors, laser distance sensors, in general, can provide more precise results, with a better reaction time, more accuracy (measuring resolution), and a lower noise level. For being able to measure differences in the quality of task execution, we used an external indoor tracking system (a VICON camera-based, high-speed tracking system [VICON, 2018]). We further used this tracking system for navigation purposes, i.e., the self-localization of the agent, and to record the exact position of the agent while moving.

This time, we used a temperature sensor instead of a distance sensor. The resulting execution of $c^v_{\text{MV-SENSOR}}$ was comparable to a kind of thermometer. If the measured temperature increased, we expected the agent to adapt its height during movement and increase it compared to the ground level. If the measured temperature decreased, we expected the agent to adapt its height accordingly and move closer to ground level.

3.4.3.2 Software and Hardware Prototypes

To deploy the necessary concepts from our Multipotent System reference architecture to SDH-prototypes, we were required to take several aspects concerning appropriate hardware into

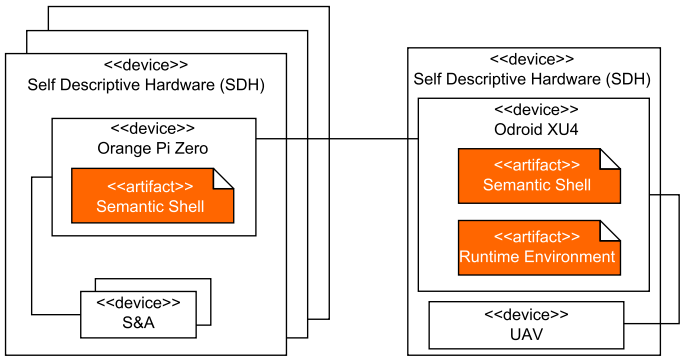


Figure 3.49: Possible deployment of the necessary software (Semantic Shell [Wanninger et al., 2018]) for different SDH in an exemplary configuration of agent α. $\mathcal{SDH}_\alpha$ consists of one SDH encapsulating an UAV and multiple other SDH encapsulating different S&A. In this setting, the Semantic Shell for the SDH encapsulating the UAV is hosted on the same hardware (an Odroid XU4 single-board computer) that also hosts the prototypical implementation of the neccessary concepts from the Multipotent System reference architecture.

account. The requirements for building flexibly configurable SDH-prototypes we identified during the development include

- appropriate physical interfaces to connect hardware internally.
- a small form and weight factor for their usage combined with another SDH that requires such, e.g., UAV.
- sufficient computing resources for the scope of functions provided by the SDH, e.g., for providing a sufficient run-time environment.
- persistent storage to store properties, capabilities, and measured values of the SDH-prototype at run-time.

To fulfill all these requirements, we used an Orange Pi Zero [CO., 2018]. Compared to other single-board computers, the form factor of the Orange Pi Zero is very small unless having sufficient performance to store and execute the capabilities of the attached sensors. Thus, the Orange Pi Zero fulfills all minimum requirements we defined above, enabling us to design a base module encapsulating the Orange Pi Zero we can use for building different SDH-prototypes (cf. Figure 3.49). In this design, we respected the special requirements arising when using reconfigurable SDH with flying agents. To mount each SDH base module to the platform hosting the agent in our experiment, we constructed a stackable 3D-printed case with magnetic connectors (cf. Figure 3.49), easing the reconfiguration within the agent's $\mathcal{SDH}_\alpha$ at run-time. For easing our experiment setup, we deployed the prototypical implementation of our Multipotent System reference architecture on the same hardware that also hosts the SDH software for the SDH encapsulating the UAV (cf. Figure 3.49). Therefore, we used an Odroid XU4 [Hardkernel, 2018] which we connected to our UAV, similar to the experiments during ScaleX 2015 and ScaleX 2016 (cf. Section 3.4.1 and Section 3.4.2). Again, we used UAV frames from *rOsewhite* [Pietzsch, 2018] combined with an Autoquad flight controller [Autoquad, 2018] to build an $\text{SDH}_{\text{AQ-I}}$ (cf. Figure 3.50a).

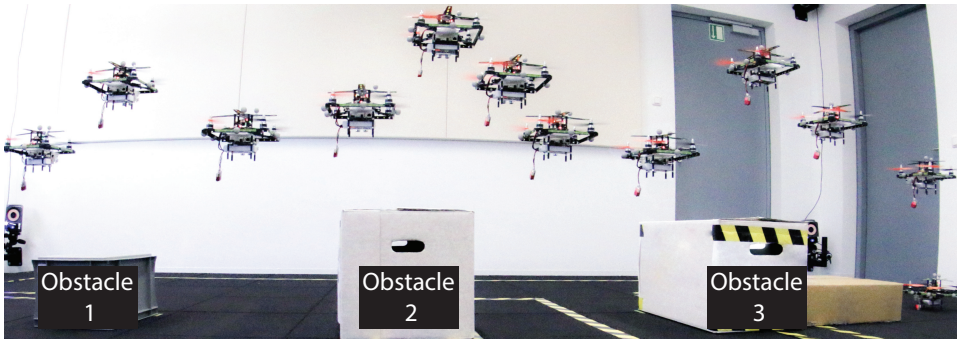
(a) SDH-prototype encapsulating an UAV (b) SDH-prototype encapsulating an ultra sonic sensor

Figure 3.50: We show the SDH-prototypes we used during the laboratory experiment. Figure 3.50a shows the SDH-prototype encapsulating an UAV designed for indoor usage in our laboratories. We equipped it with markers enabling its identification within the VICON indoor tracking system, an Odroid XU4 for hosting the SDH's software and the agent software (cf. Figure 3.49). Figure 3.50b shows a ready to use SDH-prototype encapsulating an HC SR04 ultra sonic sensor [Platform, 2021] which we used for our obstacle avoidance experiment.

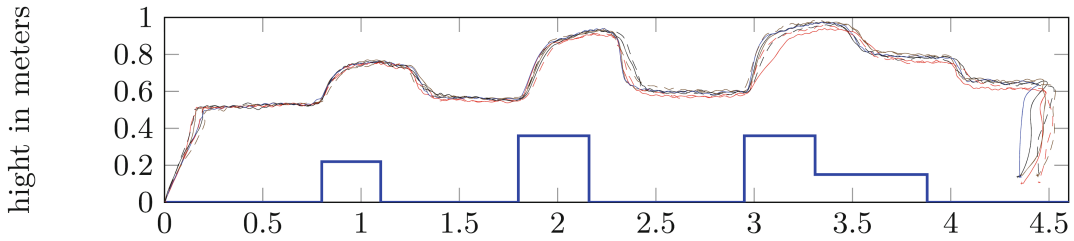
3.4.3.3 Experiment Execution and Results

We repeatedly executed our experiment with an agent configured with an $\text{SDH}_{\text{AQ-I}}$ and different types of other SDH-prototypes providing the required measuring capability for executing $c^{v}_{\text{MV-SENSOR}}$. We placed obstacles within a distance of approximately one meter, i.e., Obstacle 1 at $1m$ distance, Obstacle 2 at $2m$ distance, and Obstacle 3 at $3m$ distance. We constructed each obstacle from different materials and with different forms and sizes. With a height of approximately $0.2m$, Obstacle 2 was higher than Obstacle 1, with a height of approximately $0.4m$ With a length of approximately $1m$, Obstacle 3 was broader than Obstacle 2 (approximately $0.25m$) and Obstacle 1 (approximately $0.2m$). In Figure 3.51, we depict the execution of the task we describe in Section 3.4.3.1, performed by an agent configured with an SDH_{SR04}-prototype. We see the agent flying on different heights, adapted according to the results of the measurements performed by the SDH_{SR04}-prototype the agent uses in $c^{v}_{\text{MV-SENSOR}}$. Thereby, the agent was able to avoid each of the obstacles.

Figure 3.51a shows a time-lapse picture visualizing the movement of the agent α from a side perspective. While α moves towards its goal location by executing $c^{v}_{\text{MV-SENSOR}}$, it adapts its height according to the measurements of the respective SDH used for the experiment. In Figure 3.51b we depict the results from the experiment executed with the agent configured with an SDH_{SR04}-prototype. Values on the y-axis represent the measured height of the agent derived from our external VICON tracking system [VICON, 2018] while moving towards the goal location from left to right. We can see that while passing each of the obstacles on its way, the agent autonomously adapted its height accordingly. While the intended moving height for the task was defined at approximately $0.5m$ height, while passing the lowest Obstacle 1, the agent adapted its height to a maximum of approximately $0.8m$, ensuring to avoid collisions with the obstacle. For both higher obstacles (Obstacle 2 and Obstacle 3), the agent even adapted its height to approximately $1m$. Because Obstacle 3 had a unique form with a higher starting and a lower ending when approaching from left to right, the agent correctly reduced its moving



(a) time-lapse recording of the agent α executing $c^v_{\text{MV-SENSOR}}$ using the SDH_{SR04}-prototype, picture provided by © ISSE



(b) position measurements of the agent α during the sensor based flight

Figure 3.51: Results from the laboratory experiment. Figure 3.51a shows a time-lapse recording of the agent equipped with an SDH_{SR04}-prototype encapsulating an SR-04 sensor [Platform, 2021] while executing $c^v_{\text{MV-SENSOR}}$ on a route, facing obstacles on the ground it needs to avoid by adapting its height during moving. Figure 3.51b depicts the results of multiple experiment repetitions. The plot visualizes the moving height of the agent α while executing $c^v_{\text{MV-SENSOR}}$ from left to right towards its goal location. We derived the exact positioning of α from our external VICON tracking system [VICON, 2018].

height back to approximately $0.8m$ while passing over the obstacle. Thus, our experiment showed that we could use different flexibly configurable SDH-prototypes for instantiating the same capability (here $c^v_{\text{MV-SENSOR}}$) to modify the agents set $\mathcal{SDH}_\alpha$ of available capabilities as well as adapting the behavior of the agent while executing a capability (a distance sensor with insufficient measuring quality causes that the agent no longer could avoid the obstacles).

3.4.3.4 Conclusion

In our laboratory experiment, we were able to show the feasibility of designing and constructing prototypical SDH that influence an agent's behavior when executing virtual capabilities. The developed SDH-prototypes can store their self-description, providing common interfaces to the agent for accessing simple sensor values (e.g., different distance sensors) on the one hand or even accessing complex instructions for actuators (e.g., UAV). We demonstrated how we can use blueprints to combine different capabilities like $c^p_{\text{MV-POS}}$ and $c^p_{\text{M-DIST-G}}$, deployed on different hardware devices (cf. Figure 3.49), to more complex capabilities like $c^v_{\text{MV-SENSOR}}$.

Chapter Summary and Outlook

In this chapter, we introduced the idea of Multipotent Systems. With their design-time homogeneity and their run-time heterogeneity, we enrich with different well-established but also with new techniques of artificial intelligence, Multipotent Systems offer the flexibility and autonomy to handle SCORe missions as we describe them in Chapter 2. To reach this goal, we provide a layered reference architecture separating the functionality of agents in a PLAN LAYER, an ENSEMBLE LAYER, an AGENT LAYER, and a SEMANTIC HARDWARE LAYER.

On PLAN LAYER, we empower agents in the Multipotent System with the functionality to derive situation-specific plans from a user-designed problem definition describing the requirements of a SCORe mission. To define SCORe missions for flying ensembles, the user has access to the Multipotent System from a specific user's device offering the required interface. Because we aim at deploying Multipotent Systems to the real world, we let the user define SCORe missions with an adapted approach of Hierarchical Task Networks (HTN). HTN provide the required level of abstraction we require to avoid possible inefficiencies arising when using traditional state-space planning for the NP-hard planning problem. On ENSEMBLE LAYER, we include the necessary functionality for cooperatively working through plans derived on PLAN LAYER. Thereby, agents in the Multipotent System can autonomously form ensembles capable of working through those plans, autonomously adapt the Multipotent System's physical configuration if necessary, and autonomously coordinate the process of cooperatively working through the plans. On AGENT LAYER, we provide the agents in the Multipotent System with the functionality to participate in the executions that are initiated and coordinated on ENSEMBLE LAYER. Thereby, the AGENT LAYER serves as a link that translates coordination information provided to the ensemble into execution information the agent can process locally. We use this separation of concerns here to integrate different SO-mechanisms providing the possibility for distributing and parallelizing computation as well as scalability and robustness in execution, e.g., by exploiting swarm behavior. On SEMANTIC HARDWARE LAYER, we provide agents with the functionality for accessing hardware connected to it. Using their so-enabled self-awareness allows for agents to decide on their participation in ensembles on AGENT LAYER. Further, agents gain the possibility to execute the hardware-specific functionality when commanded by the AGENT LAYER. We encapsulate this hardware-specific functionality in different capabilities that we can vary in the course of a SCORe mission to fit the user-defined requirements when needed. To enable this possibility for physical reconfiguration, we build our approach on the concept of Self-Descriptive Hardware (SDH), offering a uniform hardware and software interface.

Besides the description of our reference architecture for Multipotent System, we describe its prototypical implementation with the Jadex Active Components Framework (Jadex). Further, we demonstrate the feasibility of deploying real robots controlled by different versions of that prototypical implementation in a set of field experiments (ScaleX 2015, ScaleX 2016) and laboratory experiments. By evaluating the results derived throughout these experiments, we show that flying ensembles implementing the Multipotent System reference architecture can provide valuable improvements to the current state in case studies we describe in Chapter 2.

In the chapters that follow (Chapters 4 to 7), we investigate in the necessary details for realizing our approach of Mission Programming for Flying Ensembles by Combining Planning with Self-Organization with Multipotent Systems. We start this description in the next chapter by focusing on designing SCORe missions and generating situation-specific plans for a Multipotent System.

Chapter 4

Ensemble Programming for Multipotent Systems

Summary. Programming goal-oriented behavior for collective systems is complex, requires high effort, and is failure-prone. In this chapter, we propose our approach for an Multi-Agent Script Programming Language for Ensembles (Maple) to deal with this challenges. Maple provides appropriate measures for task orchestration aiming at instructing Multipotent System to handle complex SCORe missions. In Maple, capabilities provided by individual agents or collectives of such serve as the instruction set an ensemble programmer can use when generating Ensemble Programs. Because the configuration of agents and their capabilities is not fixed in Multipotent Systems, we can design Ensemble Programs without taking into account the current configuration of agents. Instead of instructing concrete agents, we command instructions to abstract Planning Agents in Maple. This eases the design of Ensemble Programs and introduces new degrees of flexibility for forming ensembles to execute Ensemble Programs at run-time. It further allows us to command instructions not only to specifically identified individual agents but also to whole groups of them. The ensemble programmer so can schedule instructions to Planning Agents using the well known concepts of Hierarchical Task Networks (HTN), we extend with concepts that are necessary for their appropriate usage with ensembles. The situation-specific plans we generate using an automated planner working on so-defined HTN then encode the requirements for executing Ensemble Programs. While we focus on the possibilities for defining such requirements in this chapter, we focus on how to come by the requirements with appropriate self-organization mechanisms in Chapters 5 to 7. Thus, in this chapter we evaluate on the expressiveness of Maple by example, by applying it to our case studies and by comparing its expressiveness to that of other approaches. We further give prove of concepts by providing a reference implementation of Maple that offers a graphical front-end for designing ensemble programs and includes an implementation of our automated planner that can work with these definitions.

Publication. Contents of this chapter have been published in [Kosak et al., 2018, 2019, 2020b].

4.1 The Need for a New Approach to Ensemble Programming

In Chapter 1 and Chapter 2, we found that using mobile robots like UAV can be beneficial for many different use cases and applications. While this progress drives an ever-increasing number of agents available to a potential user, the developments for commanding and controlling such MAS/MRS have come to short. One crucial hurdle that every application needs to take before a user can profit from it is that of a proper task orchestration for the collective. Current approaches focus on this problem, e.g., with aggregate programming like Meld [Ashley-Rollman et al., 2009], and Protelis [Pianini et al., 2015] or swarm programming like Buzz [Pinciroli and Beltrame, 2016] and PaROS [Dedousis and Kalogeraki, 2018]. Unfortunately, these approaches have restrictions when the use case for the ensemble, deployed in the real world, calls for run-time task generation or heterogeneous ensembles. Other approaches aiming at such run-time task-orchestration for ensembles like Dolphin [Lima et al., 2018], TeCola [Koutsoubelias and Lalis, 2016], Voltron [Mottola et al., 2014], the approach of Gutmann and Rinner [2021], or Swarmanoid [Dorigo et al., 2013] suffer from limitations in flexibility. They are constructed for specifically composed ensembles specialized for a single-use case or do not provide proper aggregate and swarm operations, i.e., require to address individual agents directly for every single operation.

Aiming at overcoming this state of the art, we introduced the idea of Multipotent System in Chapter 3, separating the concept of capabilities from that of agents allowing for their run-time adaptation. In this chapter, we now propose our approach of a script programming language dedicated to instructing ensembles created within a Multipotent System. With the Multi-Agent Script Programming Language for Ensembles (MAPLE), we support users of Multipotent Systems with appropriate tools for expressing their needs, e.g., in the form of SCORe missions, where they can command ensembles in the Multipotent System on the individual and the collective level. Therefore, MAPLE supports the instruction of individual agents as well as instructing whole collectives by introducing the concepts of *Planning-Agent-Groups* and the concept of *Collective Capabilities*. With Planning-Agent-Groups that can instruct *all agents*, *any agent*, *a set of specific agents*, or a flexibly sized *swarms of agents* within an ensemble, we extend the flexibility the Multipotent System has for executing the instructions at run-time. As their name implies, Collective Capabilities encapsulate collective behavior, where the local interaction of individuals can provide beneficial emergent effects on the ensemble level. In MAPLE, these effects then can be included in a SCORe mission in a goal-oriented manner, e.g., for distributing an ensemble in an area with a potential field algorithm [Villa et al., 2016a] or for letting an ensemble search for the highest concentration of a parameter of interest with an adapted particle swarm optimization algorithm [Zhang et al., 2015]. In Multipotent Systems, we can alternate the collective behavior by changing the parameters of this Collective Capability in the same way we can do this for any other capabilities. We further investigate the pattern required for the execution of such an Collective Capability in Section 7.5.

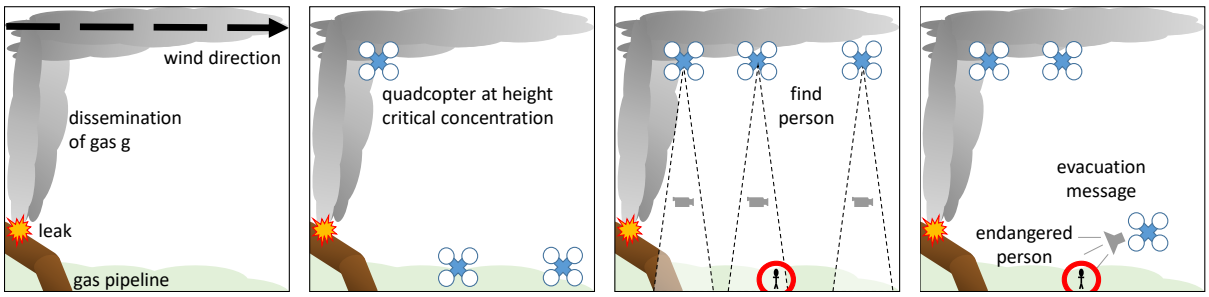
For realizing the mentioned concepts with MAPLE, we adapt the formalism of Hierarchical Task Networks (HTN) [Georgievski and Aiello, 2014], originally introduced by Erol et al. [1994], starting our extensions from the already adapted version provided by Nau [2013]. Our adaptations include new possibilities letting the mission designer schedule the control flow within ensembles by defining partial plans PART-ρ involving sequential, parallel, concurrent, conditional, and repeated instructions. Using these adapted and extended concepts of HTN, the user then can define which specific situations should trigger a respective ensemble to

execute a specific partial plan PART-ρ. We define situations in MAPLE within a world state WS that serves as a shared variables' storage. Depending on the conditions defined by the current state of WS, an automated planer then combines the user-defined partial plans PART-ρ from the HTN to a plan ρ fitting to the individual situation. Each plan ρ then encodes the requirements an ensemble $\mathcal{E}^\rho$ formed in the Multipotent System consisting of agents $\alpha \in \mathcal{E}^\rho$ with $\alpha \in \mathcal{A}_{\mathcal{MS}}$ has to fulfill afterward for executing ρ. Requirements of a plan contain the required composition of the ensemble, the necessary configuration of individual agents in that ensemble, and the cooperation pattern these agents need to follow for achieving the plan's goals. In MAPLE, we ease resolving these requirements by exploiting the main property of Multipotent System, i.e., the flexibility of agent-capability associations during run-time. Compared to other approaches, e.g., that of Koutsoubelias and Lalis [2016]; Dedousis and Kalogeraki [2018]; Mottola et al. [2014], or Dorigo et al. [2013], this flexibility enables the ensemble programmer to neglect individual agent configurations occurring at run-time during design-time. Doing so reduces the complexity when designing SCORe missions. The ensemble programmer can rely on the Multipotent System being able to generate the agent and system configurations required by the respectively generated, situation-specific plans derived from its SCORe mission descriptions in a self-organized fashion (cf. Chapter 5 providing *A Self-Organization Mechanism for Ensemble Formation* and Chapter 6 providing *A Self-Organization Mechanism for Physical Reconfiguration*). We want to note here that we explicitly do not investigate the topic of path or trajectory planning for MAS/MRS but develop an approach focusing on action planning.

In the rest of this chapter, we first define our notion of ensemble programming by using an analogy to parallel computing and work out the problem of and the challenges that lie in ensemble programming in Section 4.2. We then analyze related work in Section 4.3 concerning current approaches for the goal-oriented instructing of MAS/MRS as well as established approaches for autonomous program/plan generation working with these instructions. In Section 4.4, we then introduce our approach of MAPLE providing the relevant control structures required for ensemble programming. We combine these concepts with a planning algorithm that creates plans from the descriptions in the HTN we subsequently can transform into executable Ensemble Programs. We evaluate the expressiveness of our approach by providing minimal examples for the relevant control structures an ensemble programmer can design in an MAPLE HTN. We further evaluate the expressiveness of our approach by applying it to our case studies on *Dealing with Gas Accidents* and on *Dealing with Forest Fires*. Moreover, we compare MAPLE to other approaches for instructing collectives from the field of MAS/MRS in terms of expressiveness. In Section 4.6 we point out possible future research directions concerning our approach for ensemble program definition.

4.2 The Problem of Ensemble Programming and its Challenges

Programming collective systems and thus also programming ensembles typically turns out to be a very complex task for the programmer [Hamann et al., 2016]. As it is not easy to define what instructions an ensemble requires, i.e., what an Ensemble Program is, we analyze the requirements of such programs in the following by giving an analogy to distributed computing [Ghosh, 2014]. We perform this analysis on a simplified example originating from our case study on *Dealing with Gas Accidents* describing a typicalSCORe mission.



(a) initial scenario (b) determining height (c) find person(s) (d) evacuation message

Figure 4.1: Simplified scenario from our case study on *Dealing with Gas Accidents*. After a gas leak is detected and a gas g is exposed (Figure 4.1a), the height of a critical concentration needs to be determined (Figure 4.1b), endangered persons need to be found (Figure 4.1c), and informed with an evacuation message (Figure 4.1d)

4.2.1 Problem Analysis Performed with a Case Study

We assume a situation where a chemical accident happens in an inhabited region (cf. Figure 4.1a – Figure 4.1d). The source of a gas leak (gas g) is known, and we want an ensemble first to evaluate the height of its dissemination. Further, the ensemble should synchronize at the determined height for finding and informing potentially endangered persons (*per*). Due to weight constraints, we might have three differently configured agents α_1, α_2, and α_3 available in the system we require to cooperate in an ensemble. For determining the critical height, one agent α_1 is equipped with a gas sensor. All agents are equipped with cameras for finding persons. If the agents detect an endangered person, we want to send an evacuation message to that person using an agent equipped with a loudspeaker (α_2). The challenge for an ensemble programmer in this simple scenario, e.g., a firefighter confronted with that situation, is to program each agent in the ensemble appropriately to achieve that the ensemble as a whole accomplishes the defined mission. This job becomes way more complicated when more complex tasks require an increased amount of agents and/or capabilities [Gerkey and Matarić, 2004].

For illustration purposes, we propose a solution for solving the mission above in a distributed and asynchronous setting. We depict specialized programs for agents α_1, α_2, and α_3 in Algorithms 1 to 3, encoding the necessary coordination, capability executions, and data exchange. We differentiate between *service calls* (with an AS: prefix) that can be called by other agents (we address AS:PROCEDURE of agent α_x with α_x.PROCEDURE) and *internal procedures* that can only be called within the agent (without prefix, e.g., MOVE, MEAS, INFORM). In our code examples, we only list implementations for service calls and neglect the implementation of internal methods. We further assume that for every asynchronous service call, we can access the *caller*, e.g., for responding to the service call later. We wait for return values of services with *futures* depicted as $x \leftarrow \alpha.\text{PROCEDURE}(x)^{\downarrow}$ if we access the result x of a service call at agent α. We write $\alpha.\text{PROCEDURE}(x)^{\downarrow}$ if we wait for the service call to finish but do not require the result, and as $\alpha.\text{PROCEDURE}(x)^{\uparrow}$ if we send the service call without any interest in its result. To express parallelism, we use $(\text{PROCEDURE}_1 \| \text{PROCEDURE}_2)$ to require PROCEDURE_1 and PROCEDURE_2 to finish before continuing.

The mission encoded in the programs in Algorithms 1 to 3 that collectively form an ensemble thus is initiated by the system's user with calling AS:INIT for agent α_1 in Algorithm 1. The

Algorithm 1 AGENT α_1

```
1: procedure AS:INIT
2:   COUNTER ← 0
3:   repeat
4:     ( c^p_MV-VEL(⟨0, 0, 1⟩) ‖ g_α1 ← c^p_M-GASG )
5:   until g_α1 ≥ g_crit
6:   h_α1 ← c^p_M-POS.ALT
7:   if h_α1 < 100 then
8:     caller.WARN(h_α1)↑
9:   ( α2.MOVE(h_α1)↓ ‖ α3.MOVE(h_α1)↓ )
10:  ( per_α1 ← c^p_M-PERS ‖ α2.FIND↑ ‖ α3.FIND↑ )
11:  α1.INFORM(per_α1)↑
12: procedure AS:INFORM(per)
13:   Per ← Per ∪ per
14:   COUNTER ← COUNTER + 1
15:   if COUNTER = 3 then
16:     α2.EVAC(Per)↑
```

Algorithm 2 AGENT α_2

```
1: procedure AS:MOVE(h)
2:   c^p_MV-POS(⟨c^p_MV-POS.LON, c^p_MV-POS.LAT, h⟩)
3: procedure AS:FIND
4:   per_α2 ← c^p_M-PERS
5:   α1.INFORM(per_α2)↑
6: procedure AS:EVAC(Per)
7:   for per ∈ Per do
8:     c^p_EVAC(per)
```

Algorithm 3 AGENT α_3

```
1: procedure AS:MOVE(h)
2:   c^p_MV-POS(⟨c^p_MV-POS.LON, c^p_MV-POS.LAT, h⟩)
3: procedure AS:FIND
4:   per_α3 ← c^p_M-PERS
5:   α1.INFORM(per_α3)↑
```

Figure 4.2: Programs specifically designed for AGENTS α_1, α_2, and α_3.

agent first initializes an internal COUNTER in Line 2 required to change the executions scheduled to the agents in the ensemble in a later stage of the mission. It then starts to ascend at its current position with a velocity of 1 meters per second while measuring the concentration of the relevant gas by executing its capabilities $c^p_{\text{MV-POS}}$ and $c^p_{\text{M-GASG}}$ in parallel until the result g_{α_1} of executing $c^p_{\text{M-GASG}}$ exceeds a predefined threshold g_{crit} (cf. Line 4 and 5). Then, α_1 measures its current height h_{α_1} by using the altitude component ALT of the result from executing its capability $c^p_{\text{M-POS}}$ (cf. Line 6) an evaluates whether to inform the user (cf. Line 7 and 8). In any case, agent α_1 (as an equivalent to a master in a master/slave architecture) instructs all other participants in the ensemble (slaves in the analogy to parallel computing), i.e., agents α_2 and α_4, to also ascend to the critical height h_{α_1} determined by α_1 (cf. Line 9). Before agent α_1 can schedule the following activities within the ensemble, it waits for all other agents, i.e., α_2 and α_3, to finish this action by calling their services synchronously. Agents α_2 and α_3 consequently start ascending to the commanded height received in their respectively provided

services AS:MOVE(h) (cf. Line 1 of Algorithm 2 and Line 1 of Algorithm 3). They execute their capability $c^p_{\text{MV-POS}}$ taking their current coordinates for LAT and LON from executing $c^p_{\text{M-POS}}$ combined with the transmitted value of h for the ALT component. When agent α_1 finally registers that all other ensemble members have finished their executions, it can progress to the following command for the ensemble, i.e., determining whether endangered persons *per* are present in the monitored area. It does so by instructing all ensemble members, i.e., itself (α_1) and the other agents α_2 and α_3, to cooperatively perform this surveillance in parallel. Therefore, on the one hand, it executes its respective capability $c^p_{\text{M-PERS}}$ and, on the other hand, instructs α_2 and α_3 to do the same by calling their provided services AS:FIND without waiting for them to finish their execution (cf. Line 10). Agent α_1 then can continue its local program with an internal call of its provided service AS:INFORM(per) as soon as the result per_{α_1} from its own execution of $c^p_{\text{M-PERS}}$ is available (cf. Line 11). Because agents α_2 and α_3 do the same after finishing their local executions of $c^p_{\text{M-PERS}}$ (cf. Line 4 and 5 in Algorithm 2 and Line 4 and 5 in Algorithm 3), agent α_1 then can continue its execution when having received the results from all members of the ensemble (cf. Line 15 in Algorithm 1). Agent α_1 then can instruct agent α_2 to perform the evacuation by calling its provided service AS:EVAC(Per) (cf. Line 16). The mission is concluded by agent α_2 that executes its capability c^p_{EVAC} for every person detected by the ensemble beforehand (cf. Line 7 and 8).

A solution like we depict it in Algorithms 1 to 3 has multiple drawbacks.

1. To be executable, each program must be tailored exactly to the agent and its capabilities we want to deploy it to. If we replace one agent with another that offers different capabilities, this requires a new program for that agent.

2. Further, the only possibility to realize coordination among agents is to explicitly define it in the programs (in the example, agent α_1 also adopts this coordination role). This is unintuitive because an ensemble programmer would instead focus on the required actions in the ensemble instead of its internal interactions.

3. The same holds for data exchange between agents necessary for the correct execution of their programs. The programmer needs to explicitly define that data flow in the agents' programs achieves the expected output.

4. Another problem arises if we want to add more agents or capabilities to the ensemble. Increasing the number of agents in our example, on the one hand, would require the programmer to write a dedicated program for each new agent. On the other hand, each additional agent forces the programmer to also modify existing programs. e.g., that of the coordinator (cf. Line 9, 10 and 12 to 16 in Algorithm 1).

For coming by these drawbacks, there are two main problems to be solved. First, the ensemble programmer needs to define programs in an ensemble programming language capable of expressing all necessary operations for the ensemble. This covers the major decisions on who (which agent or which group of agents) should when and under which circumstances execute what action or sequence of actions (which of its capabilities). Second, there is a need for control and coordination structures inside the ensemble to enable the distributed execution of agent level parts of the Ensemble Program. For correct executions, some instance must be able to manage inter-agents data- and execution-flow synchronization.

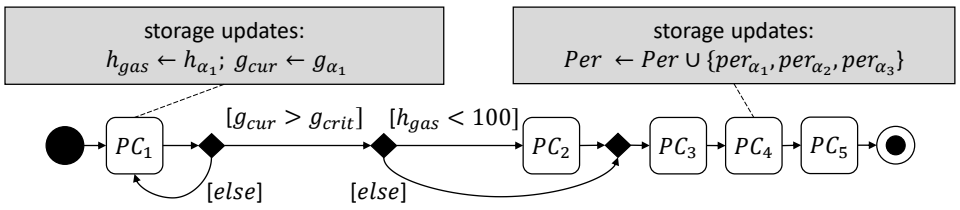


Figure 4.3: Program flow from Algorithm 1 to coordinate agents α_1, α_2, and α_3 from the ensemble formed for the mission described in Section 4.2.1. In an ensemble program, instead of addressing concrete agents with their provided services, we would like to abstractly address instructions from the instruction set $\mathcal{I}$ to ensemble processing units EPU. Each program counter PC within such an ensemble program then can hold information concerning the instructions we want to address to each EPU involved in the ensemble. Each $\text{PC}_i \in [\text{PC}_1, \text{PC}_2, \text{PC}_3, \text{PC}_4, \text{PC}_5]$ of a program then maps to an array of instructions where each entry represents the set of instructions the ensemble program requires to be executed by the respective EPU. Thus, each $\text{PC}_i = [\text{EPU}_1, \text{EPU}_2, \text{EPU}_3]$ holds instructions from $\mathcal{I}$ for all ensemble processing units relevant for the ensemble program: $\text{PC}_1 = [\{c^p_{\text{MV-VEL}}, c^p_{\text{M-GASG}}\}, \emptyset, \emptyset]$, $\text{PC}_2 = [\{caller.\text{WARN}(h_{gas})\}, \emptyset, \emptyset]$, $\text{PC}_3 = [\emptyset, \{c^p_{\text{MV-POS}}\}, \{c^p_{\text{MV-POS}}\}]$, $\text{PC}_4 = [\{c^p_{\text{M-PERS}}\}, \{c^p_{\text{M-PERS}}\}, \{c^p_{\text{M-PERS}}\}]$, $\text{PC}_5 = [\emptyset, \{c^p_{\text{EVAC}}(Per)\}, \emptyset]$.

4.2.2 Requirements of an Ensemble Programming Language

In the spirit of a procedural programming language for distributed systems that aim at multiple processors that cooperate [Bal et al., 1989], some basic ingredients need to be available to the programmer of ensemble programs for achieving complex program behavior:

- A shared storage WS for variables and their values
- an instruction set $\mathcal{I}$ consisting of instructions INSTR for modifying those variables,
- a set of ensemble processing units $\text{EPU} \in \text{EPU}$ for executing instructions from $\mathcal{I}$,
- Program flow structuring constructs for sequential, conditional, repeated, concurrent, and parallel executions with a proper syntax to express all these types of program flow and compositions of those, and
- A program control flow controlling instance PFC to coordinate the Ensemble Program's execution.

Like in other programs, controlling the executions in an ensemble requires structuring the respective Ensemble Programs, e.g., by referencing instructions with a program counter (PC). Each PC in an Ensemble Program can contain one or multiple instructions, each addressing specific EPU. Besides expressing sequential control flow in an Ensemble Program by introducing a successor concept for PC, we can explicitly use PC to express parallelism in two different forms. On the one hand, *physical parallelism* can be expressed by multiple instructions assigned to different ensemble processing units $\text{EPU}_{i \neq j}$ in the same PC (cf. PC_1 in Figure 4.3). On the other hand, *logical parallelism* can be expressed by multiple instructions assigned to the same ensemble processing unit EPU in one PC (cf. EPU_3 in Figure 4.3). Both concepts of parallelism should be supported so we can use them non-exclusively. The ensemble programmer should be able to use logical and physical parallelism in the same PC. This also calls for an appropriate fork-join concept to control the program flow in all possible parallelism ways. Besides such control operators for parallelism, the ensemble programming language also must support control operators for conditional and repeated successors of a PC. Therefore, each PC

needs to hold a table of successive PC representing the program flow graph. If this table of successors includes multiple successors for one PC, those need to be annotated with conditions excluding each other to ensure determinism. As instructions in $\mathcal{I}$ may require access to the variable storage and access to that storage may occur from different EPU, the storage needs to be shared and synchronized (e.g., with messages). With such, the ensemble programmer can define Ensemble Program like that in Figure 4.3 and even more complex ones without suffering from the drawbacks we mentioned above.

4.2.2.1 A Variable Storage WS

The programmer of an ensemble program needs to have access to the shared variable storage for writing the results of instructions and reading them back again if this is required in an Ensemble Program. Only that way we can create Ensemble Programs with complex behavior, e.g., deciding on conditional program control flow or the termination of repeated executions. The storage needs to be accessible by all EPU $\in$ EPU for saving their results when finishing an instruction from $\mathcal{I}$ if this is relevant for the program. At the same time, the instance controlling the program flow PFC must be able to read from the storage to determine the progress in the program flow. The PFC might also need to write to the storage, e.g., for aggregating results originating from multiple EPU $\in$ EPU.

4.2.2.2 An Instruction Set $\mathcal{I}$

The ensemble programming language needs to offer an API categorizing the complete supported instruction set supported by the executing system. Only this allows the ensemble programmer to access the full potential of available operations when designing Ensemble Programs. From this knowledge base, the programmer should be able to freely associate any instruction from with any EPU $\in$ EPU within the PC's included in the Ensemble Program.

4.2.2.3 Ensemble Processing Units (EPU)

The core element necessary for executing Ensemble Program are ensemble processing units EPU $\in$ EPU. These need to be able to interpret and execute program instructions from the instruction set. When the controlling instance PFC schedules instructions to an EPU $\in$ EPU because this was programmed that way in a PC, this EPU has to execute the instruction(s) and return a possibly resulting value to the controlling instance. This interaction is essential for the correct program flow of the Ensemble Program. As program instructions may contain multiple instructions to be executed logically or physically in parallel, additional requirements for EPU $\in$ EPU emerge: Ensemble processing units need to be able to execute multiple instructions from the instruction set simultaneously, keep track of their execution status, and decide for the appropriate moment for sending a response to the controlling instance, e.g., after successfully finishing the commanded instruction.

4.2.2.4 Program Flow Control (PFC)

For executing an Ensemble Program correctly, there is the need for an instance PFC for scheduling program instructions to EPU $\in$ EPU and coordinating the Ensemble Program's control flow. Therefore, a PFC needs to be aware of the Ensemble Program's control flow. While the coordination of the Ensemble Program's control flow is straightforward for simple program

instructions and sequential executions, it becomes more complicated when we need to execute conditional, parallel, concurrent, or repeated parts of the Ensemble Program. For determining which path of the Ensemble Program's control flow graph to take in case of conditional statements, the controlling instance PFC must be able to access the shared variable storage WS and evaluate the respectively relevant conditions for making correct progress in the Ensemble Program. The same holds for evaluating termination criteria in repeated executions occurring in the Ensemble Program. For recording the progress during the program execution, the controlling instance needs to be able to update (i.e., read and write) values of variables in the storage, e.g., increase counter variables. When the Ensemble Program requires parallel execution, the controlling instance PFC needs to split up the program flow, schedule it to independently acting EPU $\in$ EPU if necessary, and possibly rejoin them after their parallel execution. Thus, the controlling instance needs to be able to interact with each EPU $\in$ EPU, instruct them to execute instructions from the instruction set, and receive and correctly process their responses when available.

4.3 Related Work

Our approach aims at handling the problem of task orchestration for ensembles. Because we want to evaluate the expressiveness of our approach against other approaches focusing this topic, we investigate in their details in Section 4.5 allowing us to compare them to our approach. We base our approach on automated planning using Hierarchical Task Networks (HTN) whose concepts we extend respectively for tackling the challenges of ensemble programming we highlight in Section 4.2. Thus, we investigate in the literature concerning this topic in this section.

Planning focuses on the question of *what* has to be done to reach a goal state and *in which order*, i.e., which action sequence leads from an initial system state to the goal state [Russel and Norvig, 2014]. In this NP-complete problem domain [Russel and Norvig, 2014], researchers have made significant progress over the last decades since the first automatic planning system *STRIPS* [Fikes and Nilsson, 1971] was published. Many surveys and books on planning approaches [Alting and Zhang, 1989; Tate and Hendler, 1994; Weld, 1999; Ghallab et al., 2004; LaValle, 2006] show the long history of this problem domain. Our task orchestration approach aims to combine traditional (well-explored) planning techniques with the technique of self-organization. Thus we briefly explore the background on automated planning techniques and analyze how they are currently applied to MAS/MRS.

An adequate problem description language is essential to appropriately describe different states and describe possible solutions to the planning problem. The Planning Domain Definition Language (PDDL), currently in revision PDDL3 [Gerevini and Long, 2006], is an established technology for doing this. It enables the description of so-called "worlds"(or domains) with possible actions and their effects. Thus applying PDDL and adaptations of it require appropriate environment state abstractions and possibilities to define actions addressed to *actors* that can act in the world for modifying its state. While introducing some additional concepts into the act of planning, our approach of MAPLE aiming at task-orchestration for SCORe missions is based on the foundations of this planning language.

To find suitable plans for adequately formulated problems (e.g., with PDDL), state-space exploration [Newell et al., 1959] is the standard in many planning systems. Approaches for state-space planning enable the autonomous search for a transformation of an initial state

of the described world into a desired goal state. According to [Bonet and Geffner, 2001], approaches for searching in that state-space either follow the paradigm of forward-search or backward-search, i.e., perform progression [Hoffmann and Nebel, 2001; Haslum, 2006; Richter and Westphal, 2010] or regression [Davies and Darbyshire, 1984; Eskicioglu and Davies, 1983], and sometimes merge progression and regression for realizing a bidirectional search [Fink* and Blythe, 2005]. Forward-search approaches often exploit heuristics. An example of such heuristic is the *ignore delete lists* heuristic [Hoffmann and Nebel, 2001], which allows for monotonic progress in searching a solution in state-space by deleting adverse effects of actions (no dead ends are possible anymore). Another well known heuristic is that of task decomposition that can be used to split up a task into sub-tasks, that, solved together, have the same effect than solving the original task and relies on sub-goal independence, e.g., Abstrips [Sacerdoti, 1974], HTN [Erol et al., 1994], *SHOP2* [Nau et al., 2003]). Other approaches propose to use the heuristic of state abstraction [McDermott, 1996] to reduce the number of states in the search space by mapping multiple states to only one abstract state or recommend to generally use efficient data structures only, e.g., planning graphs in Graphplan [Blum and Furst, 1997]. These approaches for simplifying autonomous planning have shown to offer the potential of reducing the complexity for finding a plan to polynomial-time [Russel and Norvig, 2014]. Non-heuristic approaches use satisfaction planning (e.g., SATplan [Kautz and Selman, 2006]), combine Constraint Satisfaction (and Optimization) Problem (CSOP) solving with planning graphs (e.g., GP-CSP [Do and Kambhampati, 2001]), or work with partially ordered plans (e.g., Noah [Sacerdoti, 1974], Nonlin [Tate, 1976] RePOP [Nguyen and Kambhampati, 2001]) to search the space of plans instead of the space of states.

The planning problem gets even more challenging to solve when a realistic environment is assumed, where the effect of an action on the environment is uncertain [Magnenat et al., 2012]. To tackle this issue, approaches from the online [Koenig, 2001] and agent [Brafman and Domshlak, 2008] planning community were developed [Brenner and Nebel, 2009]. Instead of planning based on complete state-space knowledge, agents plan on their belief-state [Bertoli and Cimatti, 2002]. For this purpose, modeling approaches were proposed, like the widely used BDI model [Rao et al., 1995] or the MAPE-K model [Garlan et al., 2004]. If the execution of plans also needs the cooperation of different agents, planning for the distributed execution of those plans becomes indispensable [Weiss, 2013]. As planning methodologies are primarily implemented for single agents only [Awaad et al., 2014], traditional planning approaches must be adapted when dealing with distributed systems consisting of multiple agents, i.e., MAS/MRS. In such systems, planning can be accomplished either in a centralized or a decentralized fashion. Centralized approaches generate plans containing the necessary actions of all participating agents [Georgeff, 1984; Pynadath and Tambe, 2003]. There are also approaches for the interwoven execution and plan generation in MAS/MRS [Ed Durfree, 2013; Gorniak and Davis, 2007]. If otherwise, planning is to be achieved in a completely decentralized fashion, a need for guidance or policies for each agent arises to know which actions must be performed to reach a goal state cooperatively [Koenig, 2001]. This is known as the Multi-Agent Plan Coordination problem [Magnenat et al., 2009; Nissim et al., 2010; Elkawkagy and Biundo, 2011], where agents first create plans locally and then merge them into a global one. There are also approaches that map the classic solutions of centralized planning onto agent-based decentralized versions (e.g., partial-order planning in distributed HTN [Amigoni et al., 2005]). In all cases, planning in multi-agent environments needs coordination between agents [Ed Durfree, 2013], i.e., planning of individual actions and multi-agent coordination must be done together [Weiss, 2013]. All multi-agent planning techniques have in common that, finally, plans must

contain actions for every single agent of the system. When one agent's action produces an unforeseen effect, most parts of the multi-agent plan may need to be adjusted. As this is very likely in a real-world scenario most SCORe missions are located in, there is the need for further abstraction in planning, as aim for with our approach.

4.4 A Multi-Agent Script Programming Language for Ensembles (MAPLE)

Within MAPLE, we intend to hide as much complexity concerning coordination and agent interactions as possible to enable the ensemble programmer to focus its actual goal: Defining the required executions the ensemble must perform to realize the desired effect in the respective application. Thereby, we rely on the specific characteristics of Multipotent System concerning the association between agents, and their capabilities, i.e., their composition is not fixed. We use this flexibility to enable the ensemble programmer to neglect system internals during programming the ensemble. By that and in contrast to other approaches like [Hussein et al., 2014], we do not need to take the system configuration into account when creating programs for ensembles. We now describe how we can come by the challenges we enlist in Section 4.2 and propose our approach for ensemble programming. In the following, we describe how we realize the instruction set $\mathcal{I}$ defined by the possible capabilities in the system and describe how the agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ of a Multipotent System can realize the concept of $\text{EPU} \in \text{EPU}$. We further describe how we can program ensembles with a graphical programming language based on the formalism of HTN, and how we can generate the relevant information for the Multipotent System for executing so-defined programs. We thereby rely on the assumptions we made for Multipotent System in general as we describe them in Chapter 3 and further assume that capabilities once commanded for execution get executed without further planning or other disturbances. This means that for the act of planning, we assume that once planed, actions occurring in a plan are executed by the Multipotent System without requiring further complications like internal action planning or failures during the execution of actions.[1]

4.4.1 Foundations of Hierarchical Task Networks (HTN) Relevant for MAPLE

Before we propose our approach in the following, we first give a brief introduction to the concepts of HTN as they were introduced by Erol et al. [1994] and how they can be extended to be applicable in the field of artificial intelligence used in computer games like performed by Nau [2013]. Many current computer games integrate HTN planning approaches for controlling groups of Non-Player Characters (NPC) in real-time [Kelly et al., 2007, 2008; Menif et al., 2014; Neufeld et al., 2017; Vellido et al., 2020]. From our point of view, controlling such groups of NPC is very similar to controlling MAS/MRS in the real world: Groups of NPC in a computer game have a specific collective goal they want to achieve, i.e., win the game against the human player. Therefore, they may need to create plans for their actions and reason on the effects single actions have on the game's progress. In these plans, sequential and parallel actions of single NPC may be needed to achieve the collective goal. Further, groups of NPC need to

[1] We make an exception to this assumption concerning the execution of Collective Capabilities. For that, we provide a detailed description of its internal execution that inherently provides its robust execution (cf. Section 7.7).

react to possibly unexpected behavior of the environment, i.e., user inputs of the human player. Therefore, creating new plans for so-manipulated situations is often required. Because of this similarity and the successful application of HTN planning in the field of computer games, we decide on adapting the approach for HTN planning from Nau [2013] and its application within a computer game, as described by Humphreys [2013], for realizing planning for ensembles in Multipotent Systems.

When using HTN, the approach of planning is different from classical state-space planning. Instead of searching for a transition of an initial state of the world to a goal state of the world that is described as a sequence of actions (called goal-oriented planning when following the terminology of Russel and Norvig [2014]), in HTN, searching for a good solution focuses on finding a subsequent action, or short sequences of such (called partial plans), fitting best to the current situation (called task-oriented planning when following the terminology of Russel and Norvig [2014]) combined with frequent replanning. Because with that approach, an autonomous planner cannot know whether a single action moves the system towards the desired goal state in the state space, the approach of HTN relies on expert domain knowledge. While this makes the approach of HTN less general, i.e., drops the claim for problem domain independence that state-space planners make for themselves, it can increase the performance for searching for a solution [Georgievski and Aiello, 2015]. When acting in the real world, high performance in planning is urgently required. Because real-world applications introduce uncertainties that can happen during plan execution, once calculated, goal-oriented plans (like state-space planners calculate them) can quickly become obsolete as their assumptions made at planning time might spontaneously be invalidated. The so-caused need for frequent time-intense replanning makes classical state-space planning unsuitable for uncertain environments. Moreover, state-space planning relies on the assumption that all possible states of the world can be described by a finite set of predicates [Russel and Norvig, 2014]. This, of course, cannot be achieved when acting in the real world.

For coming by the difficulties arising in real-world applications, approaches for HTN aim at searching for the next advantageous *task*[2] from expert-knowledge-based domain descriptions. Because providing plans consisting of only single tasks might not be sufficient for actually improving on the system state guiding it towards the application's goal, HTN introduce two concepts for dealing with that problem we depict within Figure 4.4, taken from [Nau, 2013]. First, HTN make use of the concept of the hierarchical structuring of tasks. For realizing such hierarchies, the concept of a *task* is split into *compound tasks* and *primitive tasks*. When getting analyzed while searching for a good plan, compound tasks can be decomposed into tasks of lower complexity, i.e., other compound tasks or primitive tasks. While compound tasks do not encapsulate concrete *operations* within the goal system, those operations are then encapsulated in the primitive tasks that cannot be decomposed any further [Nau, 2013]. This approach allows for a top-down description of the problem domain, starting from the general goal defined as a compound task that can be decomposed into networks of either compound tasks (of lower complexity) or primitive tasks. Second, HTN introduce the concept of task dependencies, i.e., the idea that tasks can depend on each other, forming a network of tasks where tasks reference each other with successor relations. By interconnecting tasks (primitive or compound ones), a domain designer can define complex sequential and parallel executions as

[2] Actions are called *task* in the terminology of HTN like it was introduced by Erol et al. [1994]. Thus for the sake of consistency with literature, within this section, we also use the term *task* that way, knowing that this overloads the terminology we use in the rest of this thesis as we introduced it in Chapter 3.

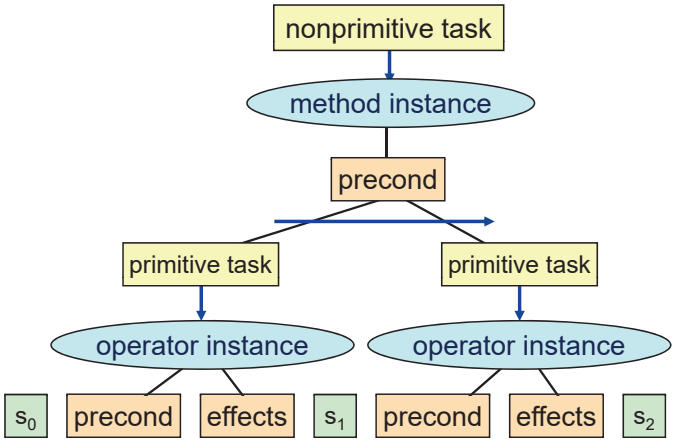


Figure 4.4: The general idea of designing HTN taken from [Nau, 2013], depicting how *compound tasks* (here called nonprimitive tasks) can be decomposed using instances of *methods*. Methods can be applied during planning if conditions formulated in the respective *precond* hold. The individual result of such decompositions are partial plans consisting of (sequences of) *primitive tasks* holding *operators* which finally encapsulate the action statements that affect the world state s. In a plan, multiple such partial plans get combined according to the successor relationships defined in the HTN.

the goal application requires. Thus, compound tasks cannot only be decomposed into partial plans consisting of single tasks that have lower complexity but also into sequences of such.

Planning on an HTN then can be described as an iterative procedure for decomposing the top-most compound task into partial plans consisting of less complex tasks, or sequences of such, until only primitive tasks remain while maintaining all successor relationships. When following the adaptations of Nau [2013] performed to the original concepts of HTN from Erol et al. [1994], situation-awareness can be achieved when creating plans by introducing multiple possibilities for decomposing a compound node called *methods*. Which of the possible decomposition, i.e., method instances (cf. Figure 4.4), the planner chooses while planning then is evaluated against the current *conditions* the system is situated in (cf. precond in Figure 4.4). The situation relevant for planning is expressed in a world state. This world state serves as a data storage for all parameters of interest relevant for the problem domain the user-designed the HTN for.

4.4.2 Capabilities as the MAPLE Instruction Set

We define the instruction set $\mathcal{I}$ to be defined by the possible capabilities $c \in \mathcal{C}$ that $\text{EPU} \in \text{EPU}$ should execute at run-time, combined with additional parameters that are necessary for the correct coordination of the Ensemble Program. The instruction set $\mathcal{I}$ is defined as very problem domain-dependent. Together with possible parameters for each $c \in \mathcal{C}$, the content of $\mathcal{I}$ provides a kind of API to the ensemble programmer. Parameters for each capability include capability-dependent data we abstractly define as p in the instruction set, i.e., the functional parameter for the capability. In addition to that, we allow also to define non-functional parameters for a capability's execution in the form of the pararmeters $s \in \text{BOOLEAN}$ (stopping criteria)

and $e \in$ BOOLEAN (external interaction), providing options to further define the capability's qualitative behavior during execution. This holds for physical capabilities $c^p_\Upsilon \in \mathcal{C}$, virtual capabilities $c^v_\Upsilon \in \mathcal{C}$ and thus also for Collective Capabilities, that, from the HTN designer perspective we adapt as ensemble programmer in MAPLE, are considered to be nothing else as a particular form of virtual capability encapsulating swarm behavior. For all instructions from $\mathcal{I}$ an ensemble programmer uses to instruct any EPU $\in$ EPU we thus require to have the parameters p, s, and e set to their respective values. Depending on the respective values set for the non-functional parameters s and e, executing an instruction then produces a result that can be further processed in the Ensemble Program directly or only with additional external guidance. We now describe the correlation of instructions, their parameters, and how and when their execution produces reusable results.

4.4.2.1 Possibilities for Internal Capability Control

The instruction's finishing type parameter $s \in$ BOOLEAN defines the instruction's execution behavior concerning the properties *how* and *when* to finish the execution of the instruction. It can either be of the type *self-finishing* if s is set to TRUE (we abbreviate with $\top$) or of the type *non-self finishing* if s is set to FALSE (we abbreviate with $\bot$). This means, if an instruction is executed with $s = \top$, the executing EPU can determine on its own when the execution of the respective capability has terminated. If, e.g., an instruction commands an EPU to move to the position $pos = \langle 1, 2, 3 \rangle$ encoded within the instruction's capability $c^p_{\text{MV-POS}}$ combined with the respective functional parameter *pos*, the EPU knows that the execution of the capability has finished when it finally reached the commanded position $\langle 1, 2, 3 \rangle$. If, instead, the ensemble programmer commands the execution of an instruction where $s = \bot$, an EPU cannot decide on the termination of the capability's execution for itself or is not allowed to do so. Instead, the capability's execution requires some external control to be terminated. This external control must either come from the program flow controlling instance PFC of the Ensemble Program the EPU is part of at run-time or by the system's user. Otherwise, e.g., for an instruction commanding an EPU to execute the capability $c^p_{\text{MV-VEL}}$ for moving with a commanded velocity encoded in the functional parameter $vel = \langle 0, 0, 1 \rangle$, the EPU executing the instruction cannot determine when to stop the execution again (no internal stopping criteria can be defined in this case). While for most capabilities setting the parameter s in the respective instruction is restricted to either be TRUE or FALSE exclusively, there may exist capabilities that are not restricted that way. A capability $c^p_{\text{M-POS-ID}}$ for measuring the position of an identifiable object *object*, e.g., may be executed with $s = \top$ to retrieve the current position of *object* once or with $s = \bot$ to track the position of *object* over time. If such capability is included in $\mathcal{C}$, i.e., also included in the instruction set $\mathcal{I}$, the ensemble programmer must decide on the capability's stopping criteria s when addressing the capability to an EPU. This decision also has an impact on whether the instruction's result can be further processed within the Ensemble Program autonomously or not. The execution of an instruction only produces a result reusable in the Ensemble Program on itself if it also finishes its execution on itself, i.e., s is set to TRUE when its execution is commanded. Otherwise, executing the capability addressed by the instruction either does not produce a result at all, e.g., when executing capability $c^p_{\text{MV-VEL}}$, or produces multiple results in a stream, e.g., when executing a measuring capability with $s = \bot$. Then, the executing EPU cannot autonomously decide which of the results from the capability's execution is the relevant one for making progress in the Ensemble Program. Then, we require further guidance for this decision performed by the controlling instance of the respective executing

ensemble PFC or performed by the system's user. Thus, the ensemble programmer needs to be aware of the impact of instructions of different types on the autonomy of Ensemble Program' execution.

4.4.2.2 Possibilities for External Capability Control

The parameter $e \in$ BOOLEAN further defines a possibility for the ensemble programmer to interact with the ensemble executing the Ensemble Program at run-time. An ensemble programmer can make use of e to require external interaction between the executing EPU and the system's user to allow the EPU to finish the execution of the respective instruction. By default, the parameter e is set to $\perp$ so that the execution is not interrupted unintentionally.

The value of parameter e then can be modified either explicitly by directly setting its value or implicitly. On the one hand, the user can set it explicitly by setting e to TRUE for a specific instruction. On the other hand, the user might set e to TRUE implicitly with its definitions for parameters of other instructions included in the Ensemble Program. This can be the case when the ensemble cannot determine on making progress in an Ensemble Program because all instructions in a PC have their respective parameter s set to FALSE (cf. Section 4.4.2.1).

In either case, if e is set to TRUE, this enforces the executing EPU and thereby the whole ensemble to halt and first continue with the Ensemble Program's execution after the commanded interaction with the user. Such behavior can be favorable if the result of executing a specific capability in an Ensemble Program has a significant impact on the subsequent control flow (cf. Section 4.4.5). If the result of executing $c^{p}_{\text{M-FIRE}}$ is TRUE, which can cause a replanning for deriving a relevant plan for the new situation, the user might want to have the final decision of whether doing so or not. Depending on the values set for non-functional parameters, we can assume that the agent we addressed the instruction to produces a result after finishing the execution of the instruction.

4.4.3 The World State as Variable Storage

The correct and goal-oriented execution of a single Ensemble Program and especially of multiple interdependent Ensemble Programs (executed by different ensembles) typically requires ensembles executing those programs to store and load their intermediate or final results. Such results can have an impact on subsequent replannings producing new plans suitable for the updated situation. Because we generate Ensemble Programs from these plans, we require such storage to produce state awareness during the execution of complex missions. We, therefore, introduce a Worldstate (WS) shared in the Multipotent System offering the possibility to do so within our approach of MAPLE. Each ensemble executing an Ensemble Program has access to the WS for reading and writing values to variables declared there. The ensemble programmer achieves the declaration and assignment of new variables in WS during the design of an HTN. The ensemble programmer can then use the values of these variables for parameters within instructions or to make decisions on the progress of conditional or repeated parts of the Ensemble Program (run-time variables usage). Further, the ensemble programmer can also make use of updated variable values in subsequent Ensemble Programs by including them in planning decisions that are evaluated when generating new plans autonomously within the Multipotent System (planning-time variables usage).

Using any variable in an HTN for the first time creates a respective entry in the WS. In general, we can modify variables using instructions of the form $variable := \langle expression \rangle$, where

variable is the respective variable in WS and *expression* can be any logical operation working on constants or values of existing variables within WS. A typical modification of a variable's value thus can be a simple assignment like $x := \bot$ or $y := \top$ or of more complex nature, e.g., $V := V \cup v$ for adding v to the already existing set of V. The ensemble programmer, in principle, can freely choose the domains of variables. However, we currently restrict them to BOOLEAN, DOUBLE, and 3D-VECTOR or sets of those for reducing the otherwise highly increased required engineering effort.

4.4.4 Planning-Agents as Ensemble Processing Units

In Multipotent Systems, we realize the concept of EPU $\in$ EPU with the different agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$. When they are correctly configured before, these agents can execute the instructions from the instruction set $\mathcal{I}$, i.e., execute capabilities $c \in \mathcal{C}$ of any type with their respectively defined functional and non-functional parameters. Because in Multipotent System, we want an ensemble programmer to be able to generate programs without taking into account the current configuration of the system, i.e., the hardware configuration $\mathcal{SDH}_\alpha$ of all agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ influencing the respectively provided set of capabilities $\mathcal{C}_\alpha$ per agent, we introduce the concept of *Planning-Agents* $\mathcal{A}^\rho$ for the act of program design. A Planning-Agent $\alpha^\rho \in \mathcal{A}^\rho$ used while defining HTN can be seen as a placeholder for an actual agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ that later, i.e., at run-time, can provide all requirements the Planning-Agent needs. The Ensemble Program generated from the plan we retrieve from automated planning using the user-designed HTN defines these requirements. Requirements concern the capabilities a respective agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ must provide for filling the placeholder of a Planning-Agent on the one hand and the interaction with other agents filling the placeholders of other Planning-Agents named in the plan on the other hand. We can derive both types of requirements from the associations between instructions and Planning-Agents the ensemble programmer creates in the HTN during design-time because these associations are also included in plans resulting from planning. We investigate how the agents in the Multipotent System collectively can assure that these requirements hold when selecting which agent should fill which placeholder (cf. the descriptions of *A Self-Organization Mechanism for Ensemble Formation* in Chapter 5 and *A Self-Organization Mechanism for Physical Reconfiguration* in Chapter 6) and when executing the Ensemble Program (cf. the routines we propose for *Executing Ensemble Programs by Using Self-Organization* in Chapter 7).

To further reduce complexity in ensemble programming and increase the flexibility for their execution during run-time, we introduce different types of Planning-Agents the programmer can use. We distinguish between Identified-Planning-Agent $\mathcal{A}^\rho_I \subset \mathcal{A}^\rho$ and Planning-Agent-Groups $\mathcal{A}^\rho_G \subset \mathcal{A}^\rho$.

4.4.4.1 Identified-Planning-Agent

Identified-Planning-Agent $\alpha^\rho{}_i \in \mathcal{A}^\rho_I$ can be reused in an Ensemble Program. Referencing to the same $\alpha^\rho{}_i$ multiple times indicates that we need the agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ filling the placeholder of $\alpha^\rho{}_i$ at run-time to execute all instructions assigned to $\alpha^\rho{}_i$. An Ensemble Program can contain multiple different Identified-Planning-Agent $\alpha^\rho{}_{1,\ldots,n} \in \mathcal{A}^\rho_I$ to require different agents $\alpha_{1,\ldots,n} \in \mathcal{A}_{\mathcal{MS}}$ to cooperate in the Ensemble Program at run-time.

Sets of Identified-Planning-Agent Sets of Identified-Planning-Agents $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$ even simplify the programming of multiple Identified-Planning-Agents. In such a set, the programmer can include multiple Identified-Planning-Agents and address them with the same instruction from $\mathcal{I}$, if this is useful for the program. When using $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$, the ensemble programmer needs to be aware of the fact that the assignment of an instruction from $\mathcal{I}$ to any Planning-Agent does not only contain the capability $c \in \mathcal{C}$ but also the capability's parameter and the non-functional parameters s and e. While using $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$ for instructing multiple ${\alpha^\rho}_i \in \mathcal{A}^\rho_I$ to execute capabilities for measuring like $c^p_{\text{M-GASG}}$ without producing any foreseeable problems, this can be different in other cases and requires further considerations from the ensemble programmer while programming. Thus, using $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$, e.g., for executing the capability $c^p_{\text{MV-POS}}$ would instruct all ${\alpha^\rho}_i \in \{\alpha^\rho_1, ..., \alpha^\rho_n\}$ to execute the capability using the same value for its functional parameter *pos*, i.e., command all agents to the same position. While this may be precisely what the ensemble programmer intends to do, the programmer also needs to be aware of possible side effects that could happen thereby in a system deployed to the real world (e.g., without a proper realization of collision avoidance in $c^p_{\text{MV-POS}}$ the instruction above might end in a crash). Thus, as stated before, we require the capabilities we use within $\mathcal{I}$ to be fully functional and enable the ensemble programmer to neglect their details necessary for execution as far as possible.

4.4.4.2 Planning-Agent-Group

Planning-Agent-Groups $\mathcal{A}^\rho_G$ are intended to not address specific agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ during run-time like we can achieve it with identified agents. Instead, using them in Ensemble Programs extends the requirements for Planning-Agents that are already included in the respective Ensemble Program. We provide three different types of Planning-Agent-Groups, the *All-Agent* ($\alpha^\rho_\forall$), the *Any-Agent* ($\alpha^\rho_\exists$), and the *Swarm-Agent* ($\alpha^\rho_{\{\substack{\text{MIN}\\\text{MAX}}\}}$).

Addressing the All-Agent $\alpha^\rho_\forall \in \mathcal{A}^\rho_G$ can instruct the whole ensemble at once. Using the All-Agent can be handy when designing Ensemble Programs, especially during the initialization and the finalization of an Ensemble Program. We can use it, e.g., for commanding the whole ensemble to get in position at the start and for returning the whole ensemble to the user at the end of a program. Like using $\{\alpha^\rho_1, ..., \alpha^\rho_n\} \in \mathcal{A}^\rho_I$, addressing instructions from the instruction set $\mathcal{I}$ to $\alpha^\rho_\forall$ extends the already defined requirements for multiple Planning-Agent at a time. However, instead of explicitly addressing the respectively associated instruction to those Identified-Planning-Agents ${\alpha^\rho}_i$ named in $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$, the All-Agent extends the requirements for all Planning-Agents included in the Ensemble Program. This holds for all ${\alpha^\rho}_i \in \mathcal{A}^\rho_I$ as well as other agents indirectly named in other Planning-Agent-Group.

Using the Any-Agent $\alpha^\rho_\exists \in \mathcal{A}^\rho_G$ can instruct one agent from the ensemble we do not need to specify further. This can be useful if we include instructions in an Ensemble Program that do not have interdependence with other instructions of any ${\alpha^\rho}_i$ named in the Ensemble Program. If, e.g., measuring the current ground temperature with $c^p_{\text{M-TEMP}}$ is relevant before starting the exemplary mission described in Section 4.2, any of three different agents occurring in a respective Ensemble Program could execute the associated capability if it has it available. The Any-Agent then generates requirements for one of these agents already included in the Ensemble Program at run-time but keeps flexibility when selecting the concrete agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$. Thus, like addressing an Identified-Planning-Agent ${\alpha^\rho}_i \in \mathcal{A}^\rho_I$, the instruction addressed

to the Any-Agent $\alpha^{\rho}_{\exists}$ should be executed by exactly one agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ during run-time. However, instead of naming one concrete place-holding $\alpha^{\rho}{}_{i}$, using $\alpha^{\rho}_{\exists}$ leaves the decision which of the agents from the ensemble executes the instruction. The use of the Any-Agent can be specifically handy when combined with the Swarm-Agent.

The Swarm-Agent $\alpha^{\rho}_{\{\substack{\text{MIN}\\\text{MAX}}\}} \in \mathcal{A}^{\rho}_{G}$ serves as an extension to the concept of the Any-Agent $\alpha^{\rho}_{\exists}$. While $\alpha^{\rho}_{\exists}$ specifies that any agent from the ensemble should execute an instruction, $\alpha^{\rho}_{\{\substack{\text{MIN}\\\text{MAX}}\}}$ defines a minimum and a maximum number of agents that should execute the referenced instruction. While we can address any instruction from $\mathcal{I}$ to the Swarm-Agent, we originally intended it to be used with the Collective Capability. When using the Collective Capability, we cannot or even do not want to determine precisely how many agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ in an ensemble should finally execute the Collective Capability. The desired emergent effect of executing a PSO encapsulated within the Collective Capability (cf. Section 7.5), e.g., can be achieved by very different sizes of the swarm. Thus, we do not want to decide on the number of participating entities at design time. Instead, using minimum and maximum bounds with $\alpha^{\rho}_{\{\substack{\text{MIN}\\\text{MAX}}\}}$ leaves this flexibility to the Multipotent System that at run-time can collectively decide on the numbers depending on the individual situation it finds itself located in. We decide to enable the possibility for restricting the number of participants in a swarm to a specific range for each usage of $\alpha^{\rho}_{\{\substack{\text{MIN}\\\text{MAX}}\}}$ as there may be minimum and maximum bounds for swarm behavior to emerge at all and stay efficient [Barca and Sekercioglu, 2013]. Because there might also be cases where an ensemble programmer can make beneficial use of assigning instructions containing other capabilities than the Collective Capability to $\alpha^{\rho}_{\{\substack{\text{MIN}\\\text{MAX}}\}}$, we thus allow this. That way, an ensemble programmer could include an instruction for, e.g., letting a minimum of 4 and a maximum of 8 agents measure temperature with $c^{p}_{\text{M-TEMP}}$ or move to the same position *pos* with $c^{p}_{\text{MV-POS}}$ if this is required.

4.4.5 Program Control Flow Structuring Using HTN Concepts

For defining the control flow of an Ensemble Program, we build on the concepts of HTN like introduced by Erol et al. [1994] and adapted by [Nau, 2013]. This allows us to express sequential and parallel executions in Ensemble Programs. We further introduce concepts allowing an ensemble programmer to express concurrent, conditional, and repeated executions. Additionally, we include the possibility to command replanning in a respective *RP* explicitly, and modify variables in the *Worldstate (WS)* with a *Planning-Time-Worldstate-Modification-Node (PWS)* or a *Runtime-Worldstate-Modification-Node (RWS)*, depending on whether we want variables to be used during run-time or during planning-time. That way, we generate a possibility for designing complex Ensemble Programs and enable an executing ensemble to generate new situation-specific Ensemble Programs through (re)planning *during run-time* autonomously. Further, designing Ensemble Programs in the form of HTN can be done in a graphical way that may also enable easier access to ensemble programming for non-technicians in the future [Bau et al., 2017]. From now on, we again speak of a *Complex-Node (CN)* and a *Primitive-Node (PN)* instead of a compound task and a primitive task (as we did in Section 4.4.1) to avoid confusion with the task concept we already use in Multipotent System (cf. Chapter 3).[3]

[3]Because the literature on planning (cf. the taxonomy used in the survey authored by Georgievski and Aiello [2014]) and the literature on Multi-Robot Task-Allocation Problem (MRTA) (cf. the taxonomy used in the survey authored by Gerkey and Matarić [2004]) use the term *task* with a partially different understanding and we combine approaches of both fields in our approach, we must introduce this redefinition.

Following the notation and taxonomy of Nau [2013], we express a possible decomposition for CN as a *Method (M)* that can be applied under certain *Condition (CON)s* that must hold in the worldstae WS.

When we address one of the elements from a HTN in the following text, we highlight it respectively by surrounding it with ⌈ and ⌋, e.g., ⌈*PN:* α^ρ*,* c⌋ for referencing a Primitive-Node PN addressing a capability c to a Planning-Agent α^ρ.

4.4.5.1 Generating Ensemble Programs from HTN

As we stated in Section 4.2, we require a concept similar to a program counter in other programs to structure the control flow of an Ensemble Program. In HTN like proposed by Nau [2013], we can find such concept in the form of PNs that encode the actions for the underlying system with an *Operator (OP)* each. When planning with an HTN, the resulting plan then includes PNs referencing those OPs. Because we extend the concepts of HTN with Runtime-Worldstate-Modification-Nodes RWS and Replanning-Nodes RP, a plan in MAPLE can also contain those nodes. We thus reference each of the nodes that occur in a plan, i.e., PNs, RWS, and RPs, with a unique program counter PC_i. Moreover, by introducing the possibility for concurrent program control flow of multiple Ensemble Programs, we must also introduce the concept of *Split-Node (SN)* we include in an Ensemble Program before splitting its control flow in multiple concurrent ones. In addition to PNs, RWS, and RPs, that occur in an HTN, also SN that first occur in plans and do not occur in HTNs receive a unique program counter.

While each RWS, RP, and SN contains instructions on the ensemble level, each OP that occurs in a PN possibly instructs different Planning-Agents $\alpha^\rho \in \mathcal{A}^\rho$ that later form the ensemble for executing the plan. Thus, we transform one plan into one ensemble level part encoding the coordination information for the ensemble on the one hand and into multiple agent level parts encoding the respective instructions from $\mathcal{I}$ referenced in OPs on the other hand (cf. Section 3.2.6), together providing the necessary information for executing the Ensemble Program at run-time. To enable the cooperation of the ensemble level part and the different agent level parts, they use the same unique program counter PC_i determined for the node in the plan they result from. We structure this and other necessary information in

- one piece of coordination information $\mathcal{CI}^\rho$ to be used by the program flow controlling instance PFC when executing the ensemble level part and
- multiple pieces of cooperation information $\mathcal{CP}^\rho$ to be used by the agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ in the ensemble when executing the respective agent level parts.

The generation of $\mathcal{CP}^\rho$ containing the relevant instructions for an agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ (cf. Table 4.2) is then a three step procedure.

1. For each unique Identified-Planning-Agent $\alpha^\rho{}_i \in \mathcal{A}^\rho_I$ referenced in any Operator OP of a Primitive-Node PN occurring in a plan ρ, we generate an individual $\mathcal{CP}^\rho_{\alpha^\rho{}_i}$. In that $\mathcal{CP}^\rho_{\alpha^\rho{}_i}$, we then reference back to the instructions originally addressed to $\alpha^\rho{}_i$ in the respective PN with the unique PC_i generated for the PN.
2. Occurrences of the Swarm-Agent $\alpha^\rho_{\{\text{MIN}\atop\text{MAX}\}} \in \mathcal{A}^\rho_G$ in Operator need special handling. Because for occurrences of the Any-Agent $\alpha^\rho_\exists$ and the Swarm-Agent $\alpha^\rho_{\{\text{MIN}\atop\text{MAX}\}}$ we cannot yet know at planning time what the concrete number of agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ participating in the respective ensemble is, we generate temporary $\mathcal{CP}^\rho_{\text{SW}}$ for each of their occurrences in any OP in a plan ρ. Depending on the allocation of the task associated to those $\mathcal{CP}^\rho_{\text{SW}}$ (cf. Chapter 5), we can then either merge the instructions from $\mathcal{CP}^\rho_{\text{SW}}$ into an existing

Table 4.1: Datatypes of the cooperation pattern information $\mathcal{CP}^\rho$ necessary to execute the agent level parts (EPU parts) and the coordination information $\mathcal{CI}^\rho$ required for executing the ensemble level part (PFC part). In $\mathcal{CI}^\rho$, $\mathcal{E}^\rho$ is a placeholder holding $\alpha^\rho \in \mathcal{A}^\rho$ until α^ρ get filled by actual agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ during run-time.

Table 4.2: cooperation pattern $\mathcal{CP}^\rho$

$\mathcal{CP}^\rho$:	MAP$\langle$PC, $\mathcal{I}_{\text{PC}}\rangle$
PC :	INT
$\mathcal{I}_{\text{PC}}$:	SET$\langle$INSTR$\rangle$
INSTR :	$c(p, s, e)$
c :	$\mathcal{C}$
p :	ANY
s :	BOOLEAN
e :	BOOLEAN

Table 4.3: coordination information $\mathcal{CI}^\rho$

$\mathcal{CI}^\rho$:	$\mathcal{E}^\rho$, PC_{TYPE}, PC_{EXP}, $\text{PC}_{\text{SUCC-MAP}}$
$\mathcal{E}^\rho$:	SET$\langle\alpha^\rho \in \mathcal{A}^\rho\rangle$
PC_{TYPE} :	MAP$\langle$PC, TYPE$\rangle$
PC_{EXP} :	MAP$\langle$PC, EXP$\rangle$
$\text{PC}_{\text{SUCC-MAP}}$:	MAP$\langle$PC, $\text{PC}_{succ}\rangle$
PC :	INT
TYPE :	ENUM{EX,STORE,PLAN,SPLIT,FINISH}
EXP :	specific instruction according to TYPE
PC_{succ} :	MAP$\langle$CONDITION, INT$\rangle$
CONDITION :	$\langle$expression defined on variables in WS$\rangle$

$\mathcal{CP}^\rho_{\alpha^\rho_i}$ or create a new $\mathcal{CP}^\rho_\alpha$ for the respective agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ that finally allocates the associated task.

3. Instructions addressing the All-Agent $\alpha^\rho_\forall$ instead are merged in any of the previously generated $\mathcal{CP}^\rho$ to be executed by the respective agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ finally participating in the ensemble $\mathcal{E}^\rho$ for that plan ρ. Thereby, the set of instructions planned for the respective PC_i in the already existing $\mathcal{CP}^\rho$ is extended by those instructions we have planned for $\alpha^\rho_\forall$ and that reference the same PC_i.

The $\mathcal{CI}^\rho$ then contains information concerning the respective ensemble $\mathcal{E}^\rho$ that needs to be coordinated, i.e., all Planning-Agents serving as placeholders for the agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ filling them at run-time in a ensemble $\mathcal{E}^\rho$, and a set of unique program counters relevant for the Ensemble Program (cf. Table 4.1). For each program counter PC_i, the $\mathcal{CI}^\rho$ further specifies

- the type PC_{TYPE} of PC_i, i.e., whether it was generated from a PN requiring the ensemble's members to execute instructions (*type* = EX), a RWS for storing some new value to a variable in WS (type = STORE), a RP requiring to execute autonomous planning for generating a new Ensemble Program (type = PLAN), or a SN for generating a concurrent control flow to that of the current Ensemble Program (type = SPLIT).

- an additional expression PC_{EXP} necessary for PC_i of type STORE, type PLAN, and type SPLIT relevant for their correct execution on ensemble level.

- a map $\text{PC}_{\text{SUCC-MAP}}$ holding information on possible succeeding PC_i depending on respective conditions.

In the following, we give respective examples for the different possibilities we have for structuring the control flow of an Ensemble Program.

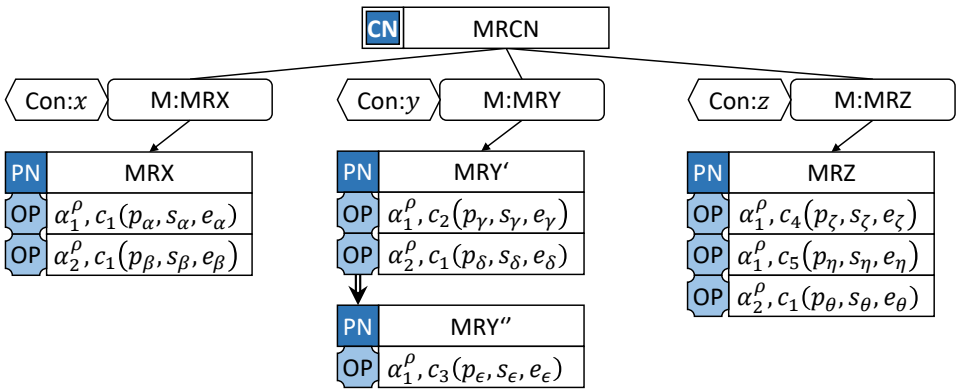


Figure 4.5: Abstract HTN schematically depicting the possibilities for designing sequential, and parallel execution for an Ensemble Program resulting from automated planning.

4.4.5.2 Defining Parallel and Sequential Control Flow

Figure 4.5 provides an abstract HTN using Operators OPs in Primitive-Nodes PNs that express sequential and parallel executions in Ensemble Programs. The different forms of control flow are included in plans generated from the possible decompositions of the topmost ⌈CN: MRCN⌋ using the different Methods ⌈M: MRX⌋, ⌈M: MRY⌋, or ⌈M: MRZ⌋. If the variables x, y, and z in WS are set to $x = \top, y = \bot, z = \bot$, we thus use the method ⌈M:MRX⌋ for decomposing ⌈MRCN⌋ because the condition ⌈CON: (x) ⌋ holds in WS and so forth.[4] For defining sequential and parallel control flow, we allow for one Primitive-Node PN to reference multiple Operators OPs at once. Each OP contains two pieces of information we use therefore. OPs in general first reference any α^ρ first $\alpha^\rho \in \mathcal{A}^\rho$, i.e., the Identified-Planning-Agent $\alpha^\rho{}_1$ referenced in the first OP in ⌈PN: MRX⌋ of the HTN in Figure 4.5. Second, each OP holds the instruction from the instruction set $\mathcal{I}$ including all parameters, i.e., p, s, and e (cf. Section 4.4.2 for their meaning).

For the sake of clarity, the Operators OPs we use in this abstract HTN all reference Identified-Planning-Agent $\alpha^\rho{}_i \in \mathcal{A}^\rho_I$. If required, we could also replace them with $\alpha^\rho_\forall$, $\alpha^\rho_\exists$, or $\alpha^\rho_{\{\substack{\text{MIN}\\\text{MAX}}\}}$ without losing the properties we describe here concerning the program's control flow.

Physical Parallelism can occur in an Ensemble Program if two or more different agents $\alpha \in \mathcal{E}^\rho$ execute instructions INSTR from the Instruction-Set $\mathcal{I}$ at the same time. We can enforce such behavior for an Ensemble Program later if, during the design of an HTN, we reference multiple Operators OPs in one Primitive-Node PN involving different Planning-Agents, e.g., different Identified-Planning-Agents $\alpha^\rho_{i\neq j}$. If we assume that, according to the Worldstate WS, we can decompose ⌈CN: MRCN⌋ from Figure 4.5 with ⌈M:MRX⌋. We thus receive a plan ρ_{MRX} containing the Primitive-Node ⌈PN: MRX⌋ that contains two Operator OPs producing such a situation. Thus, two agents at run-time represented by Identified-Planning-Agents $\alpha^\rho{}_1$ and $\alpha^\rho{}_2$ in the HTN need to execute their respectively associated capabilities in parallel: $\alpha^\rho{}_1$ must execute capability c_1 with its specific parameters $p_\alpha, s_\alpha, e_\alpha$ and $\alpha^\rho{}_2$ must execute capability c_1 with the parameters $p_\beta, s_\beta, e_\beta$. In the resulting Ensemble Program, we thus require the agents

[4] In this section, we restrict the information on planning to those pieces that are required for explaining the possibilities an ensemble programmer has for defining the program's control flow and explain the algorithm we use for planning in detail in Section 4.4.6.

$\alpha \in \mathcal{A}_{\mathcal{MS}}$ filling the placeholders of ${\alpha^\rho}_1$ and ${\alpha^\rho}_2$ to start executing the respectively simultaneously addressed capability. Therefore, we encode the associated instructions for ${\alpha^\rho}_1$ and ${\alpha^\rho}_2$ in individual $\mathcal{CP}^{\text{MRX}}_{{\alpha^\rho}_1}$ and $\mathcal{CP}^{\text{MRX}}_{{\alpha^\rho}_2}$ and reference the instructions in both pieces of information using the same program counter we use for representing the Primitive-Node ⌈PN: MRX⌋ in the respective $\mathcal{CI}^{\text{MRX}}$ (cf. PC_1 in Table 4.4 and Table 4.5). The specific coordination information $\mathcal{CI}^{MRX}$ necessary for achieving this is structured very simple, requiring only one program counter of type EX for coordinating the ensemble $\mathcal{E}^{\text{MRX}} := \{{\alpha^\rho}_1, {\alpha^\rho}_2\}$ (cf. Table 4.5). Of course, the Primitive-Node ⌈PN: MRX⌋ could involve any number of Planning-Agents referencing any instruction INSTR from the instruction set $\mathcal{I}$, creating the requirements for more than only two agents in the Ensemble Program. If one Operator OP includes other than $\alpha^\rho{}_i \in \mathcal{A}^\rho_I$, i.e., $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$ or $\alpha^\rho_\forall$ or $\alpha^\rho_{\{\substack{\text{MIN}\\\text{MAX}}\}}$, physical parallelism obviously can also be introduced in an Ensemble Program afterward, using only one instruction in one OP.

Sequential Executions in an Ensemble Program occur if the same agent $\alpha \in \mathcal{E}$ should execute multiple instructions one after the other at run-time. We can enforce that in an Ensemble Program if the same Planning-Agent α^ρ is used in different PNs that all are included in a plan. Thus, if we decompose ⌈CN: MRCN⌋ using ⌈M:MRY⌋ because the condition ⌈CON:(y)⌋ holds in the Worldstate WS (cf. Figure 4.5), we receive such a plan ρ_{MRY}. Both PNs occurring in ρ_{MRY} reference an OP associating an instruction from $\mathcal{I}$ with ${\alpha^\rho}_1$. In ⌈PN: MRY"⌋, we require ${\alpha^\rho}_1$ to execute the capability c_3 with the respective individually configured parameters $p = p_\epsilon, s = s_\epsilon, e = e_\epsilon$ only. In the previous Primitive-Node, ⌈PN: MRY'⌋, we require both, ${\alpha^\rho}_1$ and ${\alpha^\rho}_2$, to execute instructions INSTR from $\mathcal{I}$. For the Ensemble Program's control flow, this encodes two pieces of information relevant at run-time. First, we again see physical parallelism in the ⌈PN: MRY'⌋ involving both ${\alpha^\rho}_1$ and ${\alpha^\rho}_2$. Second, after finishing this parallel execution, we want ${\alpha^\rho}_1$ to execute the instruction we address to it in ⌈PN: MRY"⌋. Thus, for the control flow of the resulting Ensemble Program, we thereby design a barrier that is first falling when all instructions referenced in OPs occurring in ⌈PN: MRY'⌋ have finished. We encode this piece of information in the respective $\mathcal{CI}^{MRY}$ that includes two program counters, i.e., PC_1 and PC_2, of the type EX where we define PC_2 as the default successor of PC_1 (cf. Table 4.5). Thus before the agent $\alpha \in \mathcal{E}$ filling the placeholder of ${\alpha^\rho}_1$ at run-time is allowed to execute the instructions from the OP in ⌈PN: MRY"⌋, all $\alpha \in \mathcal{E}^\rho$ must have finished the execution of capabilities with the respectively given parameters in the program counter associated with ⌈PN: MRY'⌋. We achieve this by defining a unique program counter for ⌈PN: MRY"⌋ sequentially following the unique program counter for ⌈PN: MRY'⌋ (cf. PC_1 in Tables 4.1 and 4.4) within the respective $\mathcal{CI}^{MRY}$. Because PC_2 holds instructions for ${\alpha^\rho}_1$ only, we want the respective agent $\alpha \in \mathcal{E}$ filling the placeholder of ${\alpha^\rho}_1$ at run-time to execute the respectively named instruction (i.e., capability c_3 with its respective functional and non-functional parameters). In contrast, agent $\alpha \in \mathcal{E}$ filling the placeholder of ${\alpha^\rho}_2$ at run-time pauses meanwhile. During the phase of designing the program flow for the Ensemble Program that way, we are, of course, not restricted in the number of PNs we let sequentially follow each other or the concrete $\alpha^\rho \in \mathcal{A}^\rho$ we use in the HTN.

Logical Parallelism occurs in an Ensemble Program when the same agent $\alpha \in \mathcal{E}$ should execute more than one capability at once at run-time. We can express such by referencing the same $\alpha^\rho{}_i$ in multiple different OPs enlisted in the same PN. If we decompose ⌈CN: MRCN⌋ with ⌈M:MRZ⌋ in the HTN depicted in Figure 4.5 because the condition ⌈CON:(z)⌋ holds in the

Table 4.4: The cooperation patterns $\mathcal{CP}^{\rho}_{\alpha^{\rho}{}_i}$ generated for the different Planning-Agents $\alpha^{\rho}{}_i$ occurring in Figure 4.5 as a combination of a PC and the respective set of instructions INSTR from $\mathcal{I}$ as they were defined in the HTN for the possible plans ρ_{MRX}, ρ_{MRY}, and ρ_{MRZ}.

$\mathcal{CP}^{\rho}$	PC	$\mathcal{I}_{\text{PC}}$
$\mathcal{CP}^{\text{MRX}}_{\alpha^{\rho}{}_1}$	PC_1	$c_1(p_\alpha, s_\alpha, e_\alpha)$
$\mathcal{CP}^{\text{MRX}}_{\alpha^{\rho}{}_2}$	PC_1	$c_1(p_\beta, s_\beta, e_\beta)$
$\mathcal{CP}^{MRY}_{\alpha^{\rho}{}_1}$	PC_1	$c_2(p_\gamma, s_\gamma, e_\gamma)$
	PC_2	$c_3(p_\epsilon, s_\epsilon, e_\epsilon)$
$\mathcal{CP}^{MRY}_{\alpha^{\rho}{}_2}$	PC_1	$c_1(p_\delta, s_\delta, e_\delta)$
	PC_2	—
$\mathcal{CP}^{MRZ}_{\alpha^{\rho}{}_1}$	PC_1	$c_4(p_\zeta, s_\zeta, e_\zeta)$, $c_5(p_\eta, s_\eta, e_\eta)$
$\mathcal{CP}^{MRZ}_{\alpha^{\rho}{}_2}$	PC_1	$c_1(p_\theta, s_\theta, e_\theta)$

Table 4.5: Coordination information $\mathcal{CI}^{\rho}$ for Ensemble Programs derived from possible plans ρ_{MRX}, ρ_{MRY}, and ρ_{MRZ} generated under different conditions with the HTN from Figure 4.6.

$\mathcal{CI}^{\rho}$	$\mathcal{E}^{\rho}$-placeholder	PC	PC_{TYPE}	PC_{EXP}	PC_{succ}
$\mathcal{CI}^{\text{MRX}}$	$\mathcal{E}^{\text{MRX}} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{default \rightarrow -\}$
$\mathcal{CI}^{MRY}$	$\mathcal{E}^{MRY} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{default \rightarrow \text{PC}_2\}$
		PC_2	EX	—	$\{default \rightarrow -\}$
$\mathcal{CI}^{MRZ}$	$\mathcal{E}^{MRZ} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{default \rightarrow -\}$

Worldstate WS, we achieve a plan ρ_{MRZ} including such definitions in the $\lceil$*PN: MRZ*$\rfloor$. While we require $\alpha^{\rho}{}_2$ to only execute one instruction (cf. c_1 with $p_\theta, s_\theta, e_\theta$ within PC_1 enlisted in Table 4.4), we want $\alpha^{\rho}{}_1$ to execute two instructions at the same time (cf. c_4 with $p_\zeta, s_\zeta, e_\zeta$ and c_5 with p_η, s_η, e_η within PC_1 in Table 4.4). Because the execution of those instructions is controlled by the respective agent $\alpha \in \mathcal{E}$ exclusively at run-time, the $\mathcal{CI}^{MRZ}$ does not have any reference on this logical parallelism (cf. Table 4.5). Instead, while executing the Ensemble Program, the program flow controlling instance PFC relies on the respective agent $\alpha \in \mathcal{E}$ that fills the placeholder of the Planning-Agent at run-time to ensure the correct execution of all instructions addressed with the respective program counter in the associated $\mathcal{CP}^{\rho}$ (cf. $\mathcal{CP}^{MRZ}_{\alpha^{\rho}{}_1}$ in Table 4.4). If a plan includes such a constellation, the control flow of the respective Ensemble Program thus must trigger the execution of all capabilities for each agent $\alpha \in \mathcal{E}$ the user designs this for in the HTN.

4.4.5.3 Defining World State Modifications and Trigger Replanning

Explicit modifications of variables in WS combined with replanning can trigger the generation of new Ensemble Programs when occurring in the control flow of another Ensemble Program. By advising the ensemble to save the result of one of the instructions after finishing its execution, this result can then be used during a subsequent replanning. That way, we allow the ensemble programmer to design complex and interdependent programs and exploit the

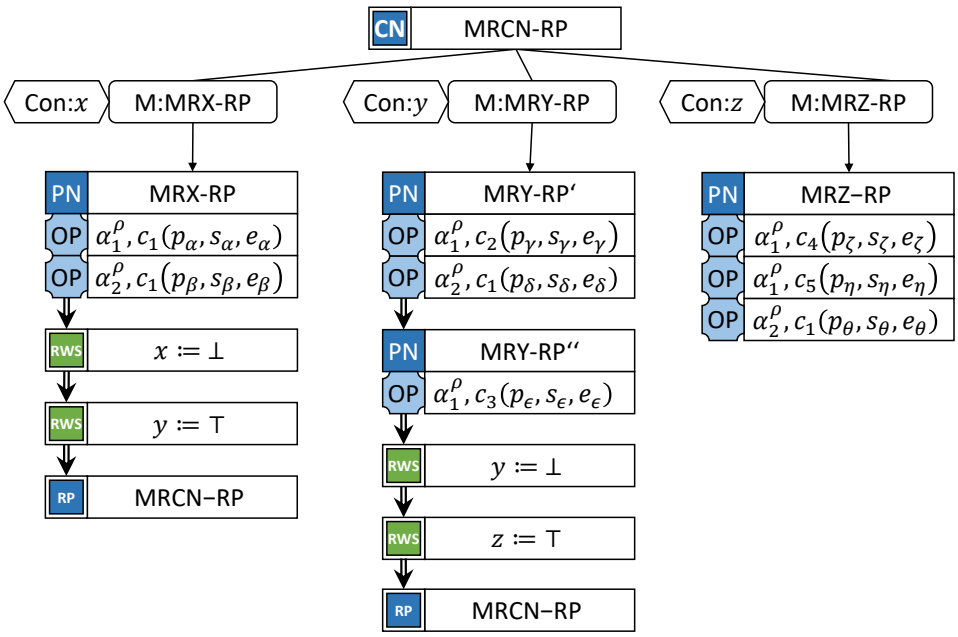


Figure 4.6: Abstract HTN depicting the possibilities for designing sequential, and parallel execution for an Ensemble Program resulting from automated planning.

situation-awareness the ensemble can generate while executing Ensemble Programs. To enable an ensemble programmer to design such functionality in HTN, we introduce the concepts of Runtime-Worldstate-Modification-Nodes RWS and Replanning-Nodes RP within MAPLE. The ensemble programmer can integrate instances of these new concepts when designing the decompositions for Complex-Nodes CNs (cf., the partial plan under $\lceil$*M:MRX-RP*$\rfloor$ in Figure 4.6). The abstract HTN in Figure 4.6 adapts the one from Figure 4.5 so that the different Ensemble Programs resulting from the possible decompositions of $\lceil$*CN: MRCN*$\rfloor$ are triggered one after the other using RPs. Again, we assume that the variables x, y, and z initially are set to $x = \top, y = \bot, z = \bot$ in WS. This causes an initial decomposition of $\lceil$*CN: MRCN*$\rfloor$ similar to that we achieve from the HTN in Figure 4.5, which we analyzed before. Compared to that, it extends the resulting plan $\rho_{MRX\text{-}RP}$ with two RWS and one RP. Both RWS and RPs encapsulate instructions for the executing ensemble to be executed in the correct order while progressing through the program's control flow. The plan $\rho_{MRX\text{-}RP}$ ends with a RP triggering the generation of a new Ensemble Program. Thus with the HTN in Figure 4.6 and the initial conditions that result in the plan $\rho_{MRX\text{-}RP}$, we create a superordinate control flow over different subsequent Ensemble Programs.

A Runtime-Worldstate-Modification-Node (RWS) or a Planning-Time-Worldstate-Modification-Node (PWS) encodes assignments of new values to variables that are defined in the Worldstate WS. We use the common form $variable := \langle expression \rangle$ (cf. Section 4.4.3). The ensemble programmer can include RWS like PWS at any desired position within a partial plan. While PWS get execvuted during planning time and thus have only influence on subsequent Ensemble Programs (cf. Section 4.4.6), RWS get executed at run-time and can influence the control flow of the current Ensemble Program.

A Replanning-Node (RP) encodes a reference towards a Complex-Node CN, where the automated planning should start to search for a new plan when the respective RP is reached while progressing the Ensemble Program's control flow. Because generating a new plan ρ_{new} also generates a new Ensemble Program requiring a new ensemble $\mathcal{E}^{\rho_{new}}$ to execute ρ_{new}, including RPs in a partial plan within an HTN requires the ensemble programmer to take additional care while designing the program. Especially when this new plan also modifies variables in WS, i.e., includes RWS (or PWS), designing partial plans that trigger RPs before modifying any variables in the WS at run-time afterward can easily cause data inconsistency.[5] Including a RP as the finishing node of such a partial plan like we illustrate in Figure 4.6 avoids such problems but still allows the ensemble programmer for creating interdependent and subsequently following Ensemble Programs, using the results derived during executing the current Ensemble Program. Thus, we currently recommend using RPs as final instruction of a partial plan only.

Integrating RWS and RPs in the Coordination Information and Cooperation Patterns Because neither RWS nor RPs include instructions for a specific agent in the ensemble, we leave their handling to the Ensemble Program control flow controlling instance PFC, i.e., the ensemble level part of the Ensemble Program. Thus, for the possible plans we derive from the HTN in Figure 4.6, we can use similar $\mathcal{CP}^{\rho}$ as we already did for those derived from the HTN in Figure 4.5, i.e., $\mathcal{CP}^{MRX}_{\alpha^{\rho}{}_i} = \mathcal{CP}^{MRX\text{-}RP}_{\alpha^{\rho}{}_i}$, $\mathcal{CP}^{MRY}_{\alpha^{\rho}{}_i} = \mathcal{CP}^{MRY\text{-}RP}_{\alpha^{\rho}{}_i}$, and $\mathcal{CP}^{MRZ}_{\alpha^{\rho}{}_i} = \mathcal{CP}^{MRZ\text{-}RP}_{\alpha^{\rho}{}_i}$ with $i \in \{1, 2\}$ That way, agents and their respective $\mathcal{CP}^{\rho}_{\alpha^{\rho}{}_i}$ are not influenced at all from RWS or RP, nor do they recognize the existence of RWS or RP in a plan.

We encode all necessary pieces of information for RWS or RPs occurring in plans in the respective $\mathcal{CI}^{\rho}$ instead. Because each node in the plan receives a unique program counter, we also reference RWS and RP with such in $\mathcal{CI}^{\rho}$. Depending on the type of the respective PC, we also include the required additional information relevant for the program flow controlling instance PFC executing the Ensemble Program. In case the program counter was generated from a RWS ($\text{PC}_{\text{TYPE}} = \text{STORE}$), we include the respective assignment as its associated expression. In case the program counter was generated from a RP ($\text{PC}_{\text{TYPE}} = \text{PLAN}$), we include the respective CN to start the replanning from as its associated expression. For the abstract HTN in Figure 4.6, we depict the $\mathcal{CI}^{\rho}$ for the different possible decompositions of $\lceil$*CN: MRCN-RP*$\rfloor$ in Table 4.6. In $\mathcal{CI}^{MRX\text{-}RP}$, e.g., we encode the RWS containing the assignment $x := \bot$ in the expression of the associated PC_2 of type STORE, the assignment $y := \top$ in the expression of a second respectively associated PC_3 of type STORE, and the RP from $\lceil$*CN: MRCN-RP*$\rfloor$ in the expression of the associated PC_4. We also adapt the successor map to respect the order defined by the ensemble programmer in the HTN, i.e., define PC_2 as default successor of PC_1, PC_3 as default successor of PC_2, PC_4 as default successor of PC_3, and PC_5 as default successor of PC_4. We consequently perform the same adaptions for $\mathcal{CI}^{MRY\text{-}RP}$ (cf. Table 4.6). Because when using the decomposition of $\lceil$*CN: MRCN-RP*$\rfloor$ from the HTN in Figure 4.6 using $\lceil$*M:MRZ-RP*$\rfloor$ results in the same plan like that of $\lceil$*CN: MRCN*$\rfloor$ using $\lceil$*M:MRZ*$\rfloor$ with the HTN depicted in Figure 4.5, there is also no difference between $\mathcal{CI}^{MRZ}$ and $\mathcal{CI}^{MRZ\text{-}RP}$.

[5] We currently do not provide a measure for handling concurrent access to WS in the current state of MAPLE and issue solving that problem to future work realized, i.e., with a decentralized but synchronized database, e.g., the CockroachDB [CockroachLabs, 2021] which we already successfully integrated experimentally in our reference architecture of Multipotent System but do not fully support within MAPLE.

Table 4.6: Possible coordination information $\mathcal{CI}^{\rho}$ for possible Ensemble Programs derived from plans generated by planning under different conditions with the HTN from Figure 4.6.

$\mathcal{CI}^{\rho}$	$\mathcal{E}^{\rho}$-placeholder	PC	PC_{TYPE}	PC_{EXP}	PC_{succ}
$\mathcal{CI}^{MRX\text{-}RP}$	$\mathcal{E}^{MRX\text{-}RP} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{default \rightarrow \text{PC}_2\}$
		PC_2	STORE	$x \leftarrow \bot$	$\{default \rightarrow \text{PC}_3\}$
		PC_3	STORE	$y \leftarrow \top$	$\{default \rightarrow \text{PC}_4\}$
		PC_4	PLAN	$\lceil$*CN: MRCN-RP*$\rfloor$	$\{default \rightarrow -\}$
$\mathcal{CI}^{MRY\text{-}RP}$	$\mathcal{E}^{MRY\text{-}RP} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{default \rightarrow \text{PC}_2\}$
		PC_2	EX	—	$\{default \rightarrow \text{PC}_3\}$
		PC_3	STORE	$y \leftarrow \bot$	$\{default \rightarrow \text{PC}_4\}$
		PC_4	STORE	$z \leftarrow \top$	$\{default \rightarrow \text{PC}_5\}$
		PC_5	PLAN	$\lceil$*CN: MRCN-RP*$\rfloor$	$\{default \rightarrow -\}$
$\mathcal{CI}^{MRZ\text{-}RP}$	$\mathcal{E}^{MRZ\text{-}RP} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{default \rightarrow -\}$

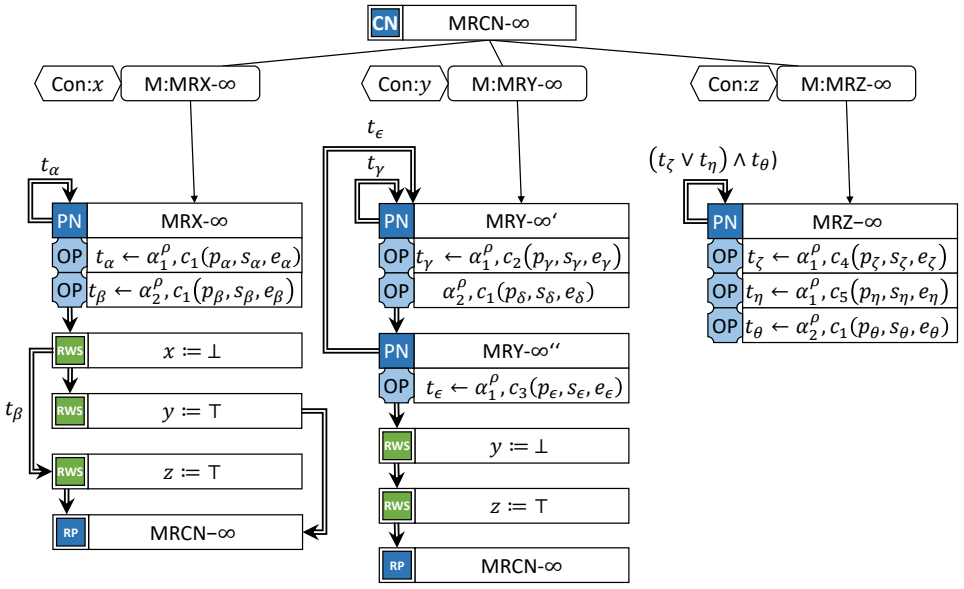


Figure 4.7: Abstract HTN depicting the possibilities for designing LOOP constructs and IF/ELSE decisions in the control flow of Ensemble Programs using conditional successors.

4.4.5.4 Defining Conditional, and Repeated Executions

Conditional and repeated executions occur in Ensemble Programs if an action in its control flow depends on the result of another action performed beforehand. The result affecting this decision can be determined by either the whole ensemble or a specific agent in the ensemble. To allow the ensemble programmer for designing such decisions with HTN, we introduce the concept of conditional successors within our approach we depict with annotated, double-lined arrows (cf. Figure 4.7). Thus, every node in the HTN cannot only have one default successor (double-lined arrow without annotation as we have already seen it in Figures 4.5 and 4.6)

Table 4.7: Possible coordination information $\mathcal{CI}^{\rho}$ for possible Ensemble Programs derived from plans generated by planning under different conditions with the HTN from Figure 4.7.

$\mathcal{CI}^{\rho}$	$\mathcal{E}^{\rho}$-placeholder	PC	PC_{TYPE}	PC_{EXP}	PC_{succ}
$\mathcal{CI}^{MRX\text{-}\infty}$	$\mathcal{E}^{MRX\text{-}\infty} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{t_\alpha \rightarrow \text{PC}_1\}, \{default \rightarrow \text{PC}_2\}$
		PC_2	STORE	$x \leftarrow \bot$	$\{t_\beta \rightarrow \text{PC}_4\}, \{default \rightarrow \text{PC}_3\}$
		PC_3	STORE	$y \leftarrow \top$	$\{default \rightarrow \text{PC}_5\}$
		PC_4	STORE	$z \leftarrow \top$	$\{default \rightarrow \text{PC}_5\}$
		PC_5	PLAN	$\lceil$*CN: MRCN-*$\infty\rfloor$	$\{default \rightarrow -\}$
$\mathcal{CI}^{MRX\text{-}\infty}$	$\mathcal{E}^{MRX\text{-}\infty} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{t_\gamma \rightarrow \text{PC}_1\}, \{default \rightarrow \text{PC}_2\}$
		PC_2	EX	—	$\{t_\epsilon \rightarrow \text{PC}_1\}, \{default \rightarrow \text{PC}_3\}$
		PC_3	STORE	$y \leftarrow \bot$	$\{default \rightarrow \text{PC}_4\}$
		PC_4	STORE	$z \leftarrow \top$	$\{default \rightarrow \text{PC}_5\}$
		PC_5	PLAN	$\lceil$*CN: MRCN-*$\infty\rfloor$	$\{default \rightarrow -\}$
$\mathcal{CI}^{\text{MRX-}\infty}$	$\mathcal{E}^{\text{MRX-}\infty} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	EX	—	$\{(t_\zeta \vee t_\eta) \wedge t_\theta \rightarrow \text{PC}_4\}$, $\{default \rightarrow -\}$

but reference a list of such successors. The annotation of those additional, i.e., conditional, successors includes a respective condition, defining under which circumstances the Ensemble Program's control flow should proceed with the respectively associated node. We require conditions for possible succeeding program counters to be mutual excluding each other, i.e., each describes a unique case. Thus, the default successor is taken if no other condition holds. That way, we can create IF/ELSE- and LOOP-structures in Ensemble Programs. In Figure 4.7, we depict an HTN further adapting the HTN from Figure 4.6 to use conditional successors in the respective decompositions of the top-most Complex-Node $\lceil$*CN: MRCN-*$\infty\rfloor$. Because the adaptations do only affect the selection of the correct program counter succeeding the respective current program counter, again all $\mathcal{CP}^{\rho}$ for possible plans derived from the HTN in Figure 4.7 stay the same like in the previously analyzed versions of the HTN from Figures 4.5 and 4.6. The only change we need to perform is an adaptation of the successor map encoded in the respective coordination information $\mathcal{CI}^{MRX-\infty}$, $\mathcal{CI}^{MRY-\infty}$, and $\mathcal{CI}^{MRZ-\infty}$.

In the decomposition of $\lceil$*CN: MRCN-*$\infty\rfloor$ using $\lceil$*M:MRX-*$\infty\rfloor$, we exemplary include two occurrences of conditional successors depending on the results of instructions executed by agents $\alpha \in \mathcal{E}^{\rho}$ filling the placeholders of $\alpha^{\rho}{}_1$ and $\alpha^{\rho}{}_2$ in $\lceil$*PN: MRX-RP*$\rfloor$ at run-time (cf. Figure 4.7). The conditional successor pointing back to itself, we annotate with t_α, is interpreted as a loop in the Ensemble Program's control flow.[6] Thus, $\mathcal{CI}^{\text{MRX}-\infty}$ includes two possible successors of PC_1, i.e., PC_1 if t_α is evaluated TRUE in WS and PC_2 as default successor (if t_α is evaluated FALSE, cf. $\mathcal{CI}^{\text{MRX}-\infty}$ in Table 4.7). While executing the Ensemble Program, thus the instructions from $\lceil$*PN: MRX-RP*$\rfloor$ are executed repeatedly from the agents $\alpha \in \mathcal{E}^{\rho}$ filling the placeholders of the Planning-Agents $\alpha^{\rho}{}_1$ and $\alpha^{\rho}{}_2$ at run-time, as long as $t_\alpha = \top$ holds in WS. The conditional successor from $\lceil$*RWS:* $x := \bot\rfloor$ pointing to the $\lceil$*RWS:* $z := \top\rfloor$ we annotate with t_β represents a IF/ELSE-structure decision combined with the default successor pointing to the RWS assigning $y := \top$. We encode this decision in the respective program counter's successor map, i.e., letting PC_4 succeed PC_2 if t_β holds in WS and otherwise continue with PC_2 (cf. $\mathcal{CI}^{\text{MRX}-\infty}$ in Table 4.7).

[6] We can directly use the result of $\alpha^{\rho}{}_1$ executing c_1 instead of first storing a variable's new value with an assignment in a RWS when we require the variable's value only temporary and in the same partial plan.

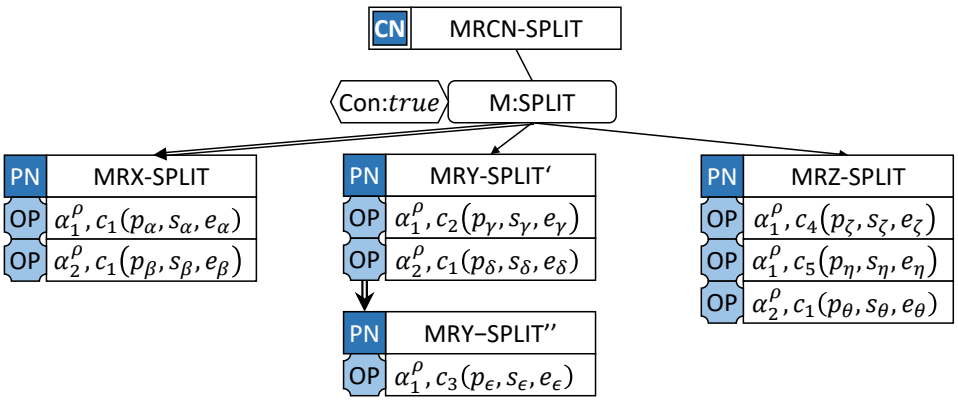


Figure 4.8: Abstract HTN depicting the possibilities for designing concurrent execution for an Ensemble Program resulting from automated planning.

That way, we can enforce different Ensemble Programs when replanning from ⌈*CN: MRCN-*∞⌋ in PC$_5$ generated from the respective ⌈*RP: MRCN-*∞⌋ contained in the HTN in Figure 4.7.

We can also nest LOOP-structures like we depict in the decomposition of ⌈*CN: MRCN-*∞⌋ using ⌈*M: MRY-*∞⌋. There, ⌈*PN: MRY-*∞*'*⌋ includes a conditional successor pointing to itself (annotated with the condition t_γ), and another conditional successor pointing from ⌈*PN: MRY-*∞*"*⌋ back to ⌈*PN: MRY*∞*'*⌋ (annotated with t_ϵ). We can express this nested and repeated program control flow desired for the Ensemble Program in the successor map of the respective $\mathcal{CI}^{MRY-\infty}$ (cf. Table 4.7).

Conditions we annotate for conditional successors can also combine the results of multiple instructions generated by different agents $\alpha \in \mathcal{E}^\rho$ filling the placeholders of the Planning-Agents at run-time. The decomposition of ⌈*CN: MRCN-*∞⌋ using ⌈*M: MRZ-*∞⌋ in Figure 4.7 depicts such with a conditional successor pointing from ⌈*PN: MRZ-*∞⌋ back to itself that logically combines the results of c_4, c_5, and c_1, i.e., $\{(t_\zeta \vee t_\eta) \wedge t_\theta \rightarrow$ PC$_4$. We can describe this desired control flow with a condition in the respective $\mathcal{CI}^{MRZ-\infty}$ (cf. Table 4.7).

4.4.5.5 Defining Concurrent Executions

Besides the possibilities for defining physical and logical parallel control flow in Ensemble Programs, we further allow the ensemble programmer to define concurrent executions. We can make use of such concept when we require multiple plans, e.g., $\rho_i \neq \rho_j$, to be executed concurrently by different ensembles, e.g., $\mathcal{E}^{\rho_i} \neq \mathcal{E}^{\rho_j}$, formed in the Multipotent System at run-time. An ensemble programmer can define concurrency in an HTN when it includes multiple suceeding partial plans in a method M used for decomposing a Complex-Nodes CN. We depict an example for such in Figure 4.8 that again represents a slightly adapted version of the HTN from Figure 4.5. Instead of decomposing ⌈*CN: MRCN*⌋ into only one of the included partial plans PART-ρ_{MRX}, PART-ρ_{MRY}, or PART-ρ_{MRZ} like we design it in the HTN we depict in Figure 4.5, decomposing ⌈*CN: MRCN-SPLIT*⌋ from the HTN in Figure 4.8 results in all those partial plans simultaneously, i.e., in PART-$\rho_{MRX\text{-}SPLIT}$ consisting of ⌈*PN: MRX-SPLIT*⌋, PART-$\rho_{MRY\text{-}SPLIT}$ consisting of ⌈*PN: MRY-SPLIT'*⌋ and ⌈*PN: MRY-SPLIT"*⌋, and PART-$\rho_{MRZ\text{-}SPLIT}$ consisting of ⌈*PN: MRZ-SPLIT*⌋. Because we define that when we let the planner autonomously generate a plan from an HTN this results in only one plan, we include the functionality for

Table 4.8: Coordination information $\mathcal{CI}^{\rho}$ for possible Ensemble Programs derived from plans generated by planning under different conditions with the HTN from Figure 4.6.

$\mathcal{CI}^{\rho}$	$\mathcal{E}^{\rho}$-placeholder	PC	PC_{TYPE}	PC_{EXP}	PC_{succ}
$\mathcal{CI}^{\text{SPLIT}}$	$\mathcal{E}^{\text{SPLIT}} = \{\alpha^{\rho}{}_1, \alpha^{\rho}{}_2\}$	PC_1	SPLIT	$\{\rho_{\text{MRY-SPLIT}}, \rho_{\text{MRZ-SPLIT}}\}$	$\{default \rightarrow \text{PC}_2\}$
		PC_2	EX	—	$\{default \rightarrow -\}$

creating concurrency at run-time. Therefore, we introduce the concept of split nodes SN that can occur in the control flow of Ensemble Programs. We reference SN with a unique program counter like we do with PN, RWS, and RP. To correctly process concurrency when executing an Ensemble Program including such SN, we require the ensemble programmer to mark one of the concurrent partial plans to be the one to continue within the current Ensemble Program in the current ensemble for differentiating this partial plan from those that should be executed concurrently by other ensembles. For the exemplary HTN in Figure 4.8, we thus reference the concurrent plans in the Ensemble Program we generate from the resulting plan ρ_{SPLIT}. We do this in the expression associated with the PC of type SPLIT which we generate for the SN (cf. Table 4.8). We further reference the marked partial plan to continue in the current Ensemble Program within the successor map of the respective SN (cf. Table 4.8). When an ensemble encounters a SN while executing the Ensemble Program at run-time, it then can inform the Multipotent System about concurrent plans encoded in the PC's expression PC_{EXP} before continuing its execution with the default PC (PC_2 in the examplary HTN depicted in Figure 4.8). The respective $\mathcal{CI}^{\rho}$ are not affected by the adaptations we perform for including concurrency and thus stay the same as we enlist them for the HTN from Figure 4.5 in Table 4.4 with $\mathcal{CP}^{\text{MRX-SPLIT}} = \mathcal{CP}^{\text{MRX}}$, $\mathcal{CP}^{\text{MRY-SPLIT}} = \mathcal{CP}^{\text{MRY}}$, and $\mathcal{CP}^{\text{MRZ-SPLIT}} = \mathcal{CP}^{\text{MRZ}}$.

4.4.6 Ensemble Program Generation Through Planning

Automated planning with an HTN then is a straightforward process working with the user-defined partial plans and control flow structuring from Section 4.4.5. Starting from the entry node NODE of the HTN that can be a Primitive-Node PN, a Planning-Time-Worldstate-Modification-Node PWS, a Runtime-Worldstate-Modification-Node RWS or a Complex-Node CN, we execute an iterative fix-point search algorithm combined with a depth-first search where we create a new plan by extending an initially empty plan ρ_i. While doing that, we can encounter any type of node in the HTN we allow in MAPLE, i.e., CNs, PNs PWS, RWS, and RPs, each with potentially multiple successor pointers. Depending on the type of the node NODE, ρ_i gets extended with the iterated procedure we describe in the following.

A) Depth-First Search in the HTN If ρ_i is empty or NODE is the only successor of the last node added to ρ_i during the last search step, we analyze the type of NODE. If NODE is a

1. PN, RWS, or RP and the node is not yet included in the current plan ρ_i, we extend ρ_i with NODE by adding NODE as a first node or as the default successor of the last node we added to ρ_i
2. PN, RWS, or RP and the node is already included in the current plan ρ_i, an infinity-loop without termination condition was created. Then we terminate the search and return ρ_i.
3. PWS, we update WS with the encoded expression without adding NODE to ρ_i

4. CN, we continue with **C)**.

We continue the search with **A)**, if NODE has only one default successor in the HTN. If NODE has more than one successor, we continue with with **B)**.

B) Successor Handling During Depth-First Search During searching for a new plan, we may encounter a NODE having multiple successors SUCC(NODE), i.e., one ore more conditional successors and one default successor. In that situation, we need to analyze all possible successors $\text{NODE}_{\text{SUCC}} \in \text{SUCC}(\text{NODE})$ including all their respective further successors $\text{SUCC}^*(\text{NODE})$. We thus analyze for each successor $\text{NODE}_{\text{SUCC}} \in \text{SUCC}(\text{NODE})$ whether

1. ρ_i already contains $\text{NODE}_{\text{SUCC}}$, i.e., NODE is included in $\text{SUCC}^*(\text{NODE}_{\text{SUCC}})$, we need to decide whether to add the $\text{NODE}_{\text{SUCC}}$ again or not: If for reaching NODE by traversing $\text{SUCC}^*(\text{NODE}_{\text{SUCC}})$ during search
   a) a Complex-Nodes occurred (cf. **C)**), we add $\text{NODE}_{\text{SUCC}}$ again to ρ_i and point towards it with a respectively annotated condition (or mark it as default successor). In this case we rely on different decompositions of the respective Complex-Nodes during planning for terminating the search eventually, e.g., by modifications performed on the variables in the Worldstate WS using Planning-Time-Worldstate-Modification-Nodes PWS (we can see that structure as a kind of FOR-LOOP-construct, evaluated during planning time).
   b) otherwise we do not include $\text{NODE}_{\text{SUCC}}$ again in ρ_i but create an additional reference from NODE back to $\text{NODE}_{\text{SUCC}}$ annotated with the respective condition (or mark it as default successor).
2. ρ_i does not contain $\text{NODE}_{\text{SUCC}}$, we add $\text{NODE}_{\text{SUCC}}$ to ρ_i and create a reference from NODE to $\text{NODE}_{\text{SUCC}}$ annotated with the respective condition (or mark it as default successor).

We proceed for each added node $\text{NODE}_{\text{SUCC}}$ with **A)** until no more new nodes can be found.

C) Decomposition of CN Because we require a final plan to consist only of PNs, RWS, PWS, and SN, which we can reference with program counters for making them executable in an Ensemble Program at run-time, we need to decompose all CNs occurring during the search. When we find such during **A)**, we refer to it as NODE_{CN} and

1. decompose NODE_{CN} by analyzing which of its methods M holds within WS,
2. if the method that applies for NODE_{CN} within the current state of WS points at
   a) one partial plan PART-ρ_1, then
      i. we initialize a new temporary empty plan ρ_i^{TEMP},
      ii. recursively start with **A)**, using the first node of PART-ρ_1 as first node for ρ_i^{TEMP} and then continue with **C)**3.
   b) multiple partial plans, including one default partial plan PART-ρ_{DEF} and multiple concurrent partial plans PART-$\rho_{1,\ldots,n}$, then
      i. we initialize a new temporary empty plan ρ_i^{TEMP}
      ii. add a SN as first node to ρ_i^{TEMP},
      iii. create a new plan $\rho_i^{\text{TEMP-DEF}}$ for the default partial plan PART-ρ_{DEF} marked in the method,
      iv. create a new plan $\rho_i^{\text{TEMP-1,\ldots,TEMP-N}}$ for PART-$\rho_{1,\ldots,n} \neq$ PART-ρ_{DEF}

v. recursively start with **A)** again for each PART-$\rho \in \{\text{PART-}\rho_{1,\dots,n} \cup \text{PART-}\rho_{\text{DEF}}\}$, using the respective first node of PART-ρ as NODE.

vi. integrate the temporary plans $\rho_i^{\text{TEMP-DEF}}$ and $\rho_i^{\text{TEMP-1,...,TEMP-N}}$ into ρ_i^{TEMP} after finishing each recursive call of **A)** by

A. adding a default successor to the SN referencing the first node of PART-ρ_{DEF}

B. adding a concurrent successor to the SN referencing the respective first node of all concurrent plans $\rho_i^{\text{TEMP-1,...,TEMP-N}}$ and then continue with **C)**3.

3. integrate ρ_i^{TEMP} into ρ_i after finishing each recursive call at the position in ρ_i we encountered NODE_{CN} by adjusting the respective successor pointers.

4.5 Evaluation

In Section 4.3, we found that besides its expressiveness the benefit of planning with HTN consisting of expert-knowledge-based partial plans is the efficiency we gain for the act of planning. Compared to state-space planning, the time required for planning with HTN is low, at least for all HTN not involving massive possible decisions and partial plans [Humphreys, 2013], and thus we can neglect it in our evaluations.

Thus, in our evaluations on planning with MAPLE, we do not focus on the efficiency of our planning algorithm from Section 4.4.6 but instead investigate the expressiveness of our approach. In the following, we, therefore, give examples originating from our reference implementation of MAPLE. We thereby aim at evaluating the possibilities we provide for combining our extended concepts of HTN with the planning algorithm we use in MAPLE. For illustrating the planning process, we focus on the different single aspects we provide in MAPLE first. Second, we evaluate the expressiveness of MAPLE by applying it to three case studies: our example from Section 4.2, a newly introduced seeding robot scenario, and the case study of *Dealing with Forest Fires*, each using the necessary aspects of our approach combined within our reference implementation. Third, we evaluate the expressiveness of our approach MAPLE by comparing it to that of other current approaches aiming at ensemble programming and task-orchestration for ensembles.

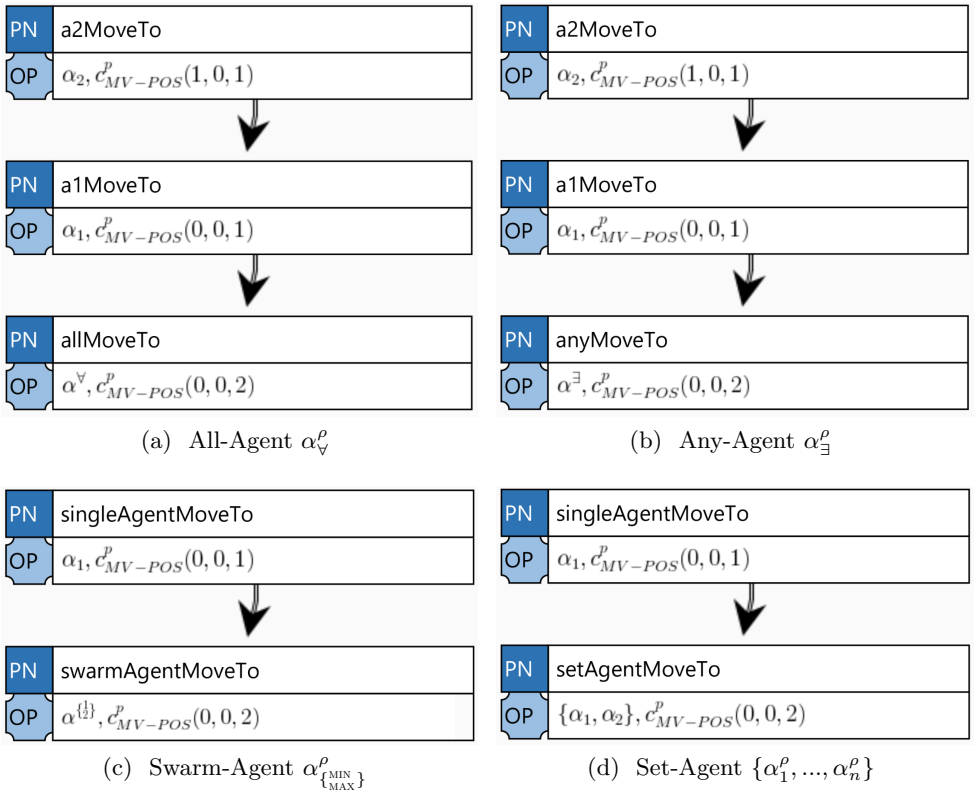
Because all figures depicting HTNs and plans in this section stem from our prototypical reference implementation of Multipotent Systems we integrated MAPLE with, we need to perform a slight switch in representation for technical reasons here: We represent decompositions of CNs as single-lined arrows with non-filled tips (instead of single-lined arrows with filled tips). Deciding for this minor discrepancy allows us to complement the evaluations concerning expressiveness we provide in this section with showcasing their execution using our reference implementation in Chapter 7 focusing *Executing Ensemble Programs by Using Self-Organization*.

4.5.1 Evaluating the Expressiveness of MAPLE Using Examples

We first evaluate the expressiveness of our approach by providing examples making use of all elements we describe in Section 4.4.

4.5.1.1 Using Planning-Agent-Groups

To demonstrate the expressiveness concerning different types of planning agents we support with MAPLE, we give some minimalistic examples in the following for each type (except from



(a) All-Agent $\alpha^\rho_\forall$ (b) Any-Agent $\alpha^\rho_\exists$

(c) Swarm-Agent $\alpha^\rho_{\{^{\text{MIN}}_{\text{MAX}}\}}$ (d) Set-Agent $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$

Figure 4.9: Four different examplary HTN using the different types of Planning-Agents $\mathcal{A}^\rho$ we support in our approach. Figures exported from our prototypical reference implementation (cf. Section 3.3.5.1).

Identified Planning Agents ${\alpha^\rho}_i \in \mathcal{A}^\rho_I$, which we already use in all of our other following examples). To reduce complexity in these examples, we avoid using CNs, PWS, RWS, RPs, and use only default successors. To demonstrate the possibility for expressing these other functionality, we present them in isolated examples later. Thus because of the lack of conditional decisions, plans resulting from automated planning using the HTN we depict in Figure 4.9 do not alter in their form when compared to the HTN they originate from. Figure 4.9 thus shows the exemplary usage of the Set-Agent $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$, the All-Agent $\alpha^\rho_\forall$, the Any-Agent $\alpha^\rho_\exists$, and the Swarm-Agent $\alpha^\rho_{\{^{\text{MIN}}_{\text{MAX}}\}}$.

In Figure 4.9a, we first instruct the Planning-Agent ${\alpha^\rho}_2$ to execute the capability $c^p_{\text{MV-POS}}$ to the coordinate $\langle 1,0,1\rangle$ within the only OP listed in the Primitive-Node ⌈*PN: a2MoveTo*⌋. We then instruct another Planning-Agent ${\alpha^\rho}_1$ to execute the capability $c^p_{\text{MV-POS}}$ to a different coordinate $\langle 0,0,1\rangle$ in a respective default successor ⌈*PN: a1MoveTo*⌋. In the following default successor ⌈*PN: allMoveTo*⌋, we make use of the *All-Agent* by instructing $\alpha^\rho_\forall$ to execute the capability $c^p_{\text{MV-POS}}$ to a third coordinate $\langle 0,0,2\rangle$. Thus, the ensemble programmer encodes its goal in the HTN that the two agents in an ensemble executing the plan at run-time should first, one after the other, move to two different positions and then rendezvouz at a third position. If we included more Planning Agents in PNs before using $\alpha^\rho_\forall$ in a last PN, we would also command all of them to the rendezvouz position.

In Figure 4.9b, we use a very similar HTN compared to that in the HTN in Figure 4.9a.

While ⌈PN: *a2MoveTo*⌋ and ⌈PN: *a1MoveTo*⌋ include the very same instructions for the same Planning-Agent, we command the *Any-Agent* $\alpha^{\rho}_{\exists}$ instead of the All-Agent $\alpha^{\rho}_{\forall}$ in the last Primitive-Node ⌈PN: *anyMoveTo*⌋. Thus, the ensemble programmer encodes its goal in the HTN, that after the agents of the ensemble filling the placeholders of ${\alpha^{\rho}}_1$ and ${\alpha^{\rho}}_2$ at run-time have reached their commanded positions, only one of them (i.e., any agent $\alpha \in \mathcal{E}^{\rho}$) should subsequently move to the coordinate $\langle 0,0,2\rangle$ commanded in the Operator ⌈OP: $\alpha^{\rho}_{\exists}$, $c^{p}_{MV\text{-}POS}$ $\langle 0,0,2\rangle$⌋ instead of all agents. Because the HTN does not define whether this should be ${\alpha^{\rho}}_1$ or ${\alpha^{\rho}}_2$ explicitly, in different executions of the plan the instruction of ⌈OP: $\alpha^{\rho}_{\exists}$, $c^{p}_{MV\text{-}POS}$ $(0,0,2)$⌋ thus could be executed by different agents from $\mathcal{E}^{\rho}$.

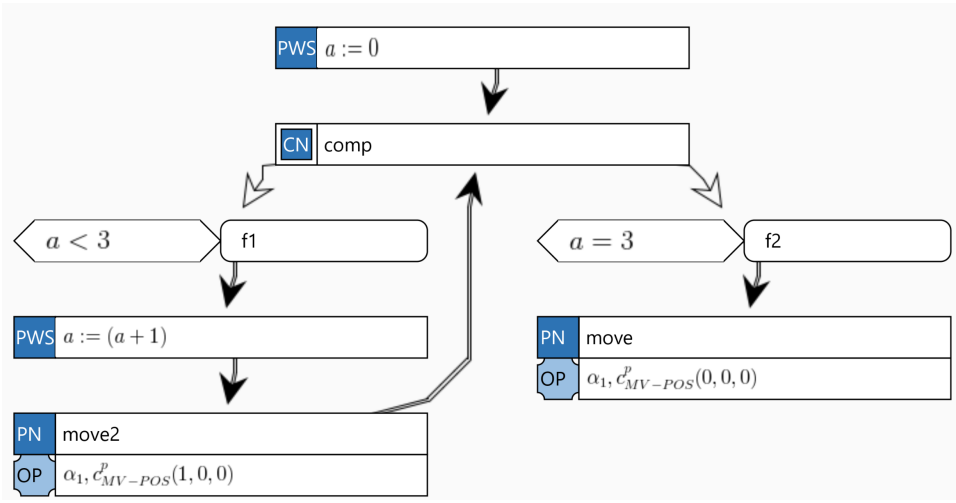
In Figure 4.9c, we use the *Swarm-Agent* to define behavior for the Ensemble Program similar to that of the two examples before. First, we again command ${\alpha^{\rho}}_1$ to execute the capability $c^{p}_{\text{MV-POS}}$ to move to the coordinate $\langle 0,0,1\rangle$ in the only OP listed in the Primitive-Node ⌈PN: *singleAgentMoveTo*⌋. Subsequently, we command at least one and a maximum of two agents from the executing ensemble to execute $c^{p}_{\text{MV-POS}}$ to move to the coordinate $\langle 0,0,2\rangle$ using the Swarm-Agent in ⌈OP: ${\alpha^{\rho}}_{\{1\}_2}$, $c^{p}_{MV\text{-}POS}$ $(0,0,2)$⌋ enlisted in ⌈PN: *swarmAgentMoveTo*⌋. Thus, by designing the HTN that way, the executing ensemble can consist of one, two, or three agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ depending on their concrete availability during run-time. If the same agent α_1 filling the placeholder of ${\alpha^{\rho}}_1$ is the only agent in the swarm in ⌈PN: *swarmAgentMoveTo*⌋, the ensemble has the size of one. If there is another agent α_2 participating in the swarm, the ensemble has a size of two. If the agent filling the placeholder of the Identified-Planning-Agent ${\alpha^{\rho}}_1$ is no part of the swarm, there can either be one or two other agents participating in the swarm, i.e., the ensemble has a size of two or three. Thereby, the ensemble programmer can leave as much flexibility to the system to self-organize during run-time as desired for the respective use case.

In Figure 4.9d, we use the *Set-Agent* commanding the Identified-Planning-Agents ${\alpha^{\rho}}_1$ and ${\alpha^{\rho}}_2$ to execute the capability $c^{p}_{\text{MV-POS}}$ for moving to the coordinate $\langle 0,0,2\rangle$ in the Operator ⌈OP: $\{{\alpha^{\rho}}_1, {\alpha^{\rho}}_2\}$ $c^{p}_{MV\text{-}POS}$ $(0,0,2)$⌋ listed in the Primitive-Node ⌈PN: *setAgentMoveTo*⌋ following the Primitive-Node ⌈PN: *singleAgentMoveTo*⌋ listing the same OP as in the last example (cf. Figure 4.9c). Thus here, the ensemble programmer explicitly requires the agetn filling the placeholder of ${\alpha^{\rho}}_1$ at run-time to execute both instructions and further sets the required ensemble size to a fixed number of two by involving an additional Identified-Planning-Agent ${\alpha^{\rho}}_2$.

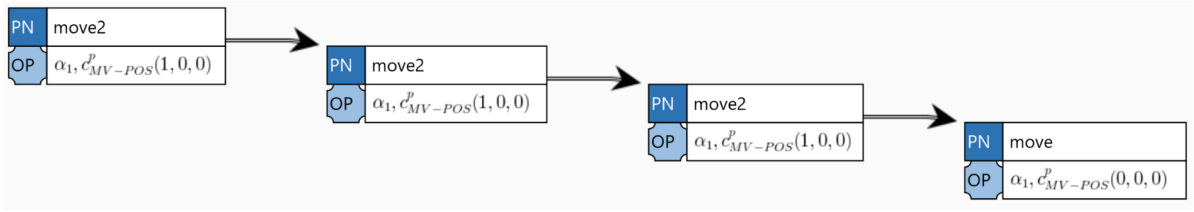
Thus, we demonstrate by example that we provide a maximum expressiveness to the ensemble programmer for implicitly and explicitly defining the roles in a plan with the different types of Planning-Agents we can address instructions to in MAPLE.

4.5.1.2 Using Variables and Decomposition

In Figure 4.10, we give an exemplary HTN involving Planning-Time-Worldstate-Modification-Nodes PWS and Complex-Nodes CNs for realizing an iterative decomposition while creating a plan (cf. the description of the planning algorithm in Section 4.4.6). First, the variable a in the Worldstate WS is set to 1 in the Planning-Time-Worldstate-Modification-Node ⌈PWS: $\{a := 1\}$⌋. The Complex-Node ⌈CN: *comp*⌋ uses this variable a in the conditions for applying different methods, i.e., ⌈CON: $(a < 3)$⌋ for ⌈M: *f1* and ⌈CON: $(a < 3)$⌋ for ⌈M: *f2*⌋. Because we later again modify the variable a, in the ⌈PWS: $a := \{a + 1\}$⌋ we receive a plan requiring 3 times ⌈PN: *move2*⌋ before finally executing ⌈PN: *moveTo2*⌋ in the resulting plan. This comes from the iterative decomposition of the Complex-Node ⌈CN: *comp*⌋ we create by referencing a



(a) User-defined HTN using PWS and Decomposition.



(b) Resulting plan from planning with the HTN from Figure 4.10a.

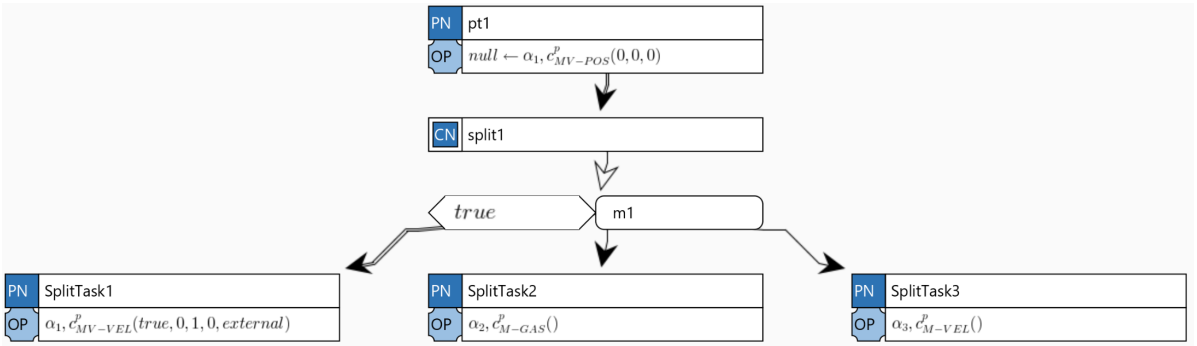
Figure 4.10: A HTN involving PWS for realizing an iterative decomposition in Figure 4.10a and the resulting plan in Figure 4.10b. Figures exported from our prototypical reference implementation (cf. Section 3.3.5.1).

Primitive-Node PN predecessing ⌈*CN: comp*⌋ and modifying WS during planning in one of the decompositions of the Complex-Node ⌈*CN: comp*⌋. Thus, instead of explicitly defining this plan in a single partial plan consisting of all required nodes, the ensemble programmer can exploit the expressiveness of MAPLE to define iterative executions in an Ensemble Program if desired and design FOR-LOOP-like constructs during programming.

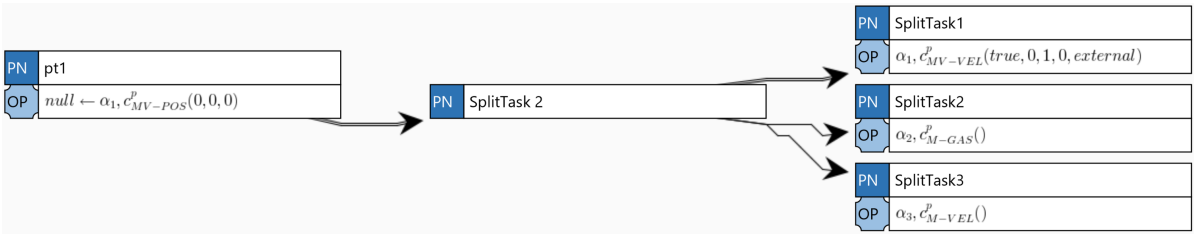
4.5.1.3 Using Control Structures

The ensemble programmer can make use of different patterns in HTN, each expressing different control structures in the Ensemble Programs resulting from the individual plans. In the following, we demonstrate these possibilities for evaluating the expressiveness of our approach.

Concurrent Partial Plans In Figure 4.11, we depict the possibility for the ensemble programmer to express the need for multiple concurrent plans. In the HTN in Figure 4.11a, after an initial Primitive-Node ⌈*PN: pt1*⌋ instructing the Identified-Planning-Agent $\alpha^{\rho}{}_1$ to execute the capability $c^p_{\text{MV-POS}}$ for moving to the coordinate $\langle 0,0,0\rangle$ in the listed Operator OP, we find the Complex-Node ⌈*CN: split1*⌋ as its default successor during planning. Because ⌈*CN: split1*⌋ offers only one Method ⌈*M: m1*⌋ associated with the Condition ⌈*CON: true*⌋ relevant for decomposition, we use this method. By providing more than one subordinate partial plan, this



(a) A HTN containing concurrent partial plans.



(b) Resulting plan from planning with the HTN from Figure 4.11a.

Figure 4.11: A HTN and its respective plan containing a control structure for the Ensemble Program producing concurrent plans. Figures exported from our prototypical reference implementation (cf. Section 3.3.5.1).

decomposition first creates a Split-Node ⌈PN: *SplitTask3*⌋ in the plan (cf. Figure 4.11a).[7] Then we include all partial plans listed in the Method ⌈M: *m1*⌋ as successors of the Split-Node ⌈PN: *SplitTask3*⌋ and annotate the respectively marked successor as the default (cf. the Primitive-Node ⌈PN: *s1*⌋ in Figure 4.11). We mark all other successors (i.e., the Primitive-Nodes ⌈PN: *s2*⌋ and ⌈PN: *s3*⌋) as concurrent successors of ⌈PN: *SplitTask 3*⌋ and thus also of ⌈PN: *pt1*⌋ continuing the original plan.

Using that pattern in a HTN during design, the ensemble programmer thus can express concurrent parallelism if this is required for its use case. As stated before, when designing such concurrent parallelism, the ensemble programmer needs to take special care when using and modifying variables defined in the Worldstate WS. For avoiding inconsistencies of variable assignments, appropriate locking mechanisms are required to avoid typical issues that can happen when concurrently accessing shared storage like read-before-write, write-before-write, and write-before-read access. Distributed and synchronized databases like the CockroachDB [CockroachLabs, 2021] can provide a possible solution to this issue.

Repeated Executions using RWS in Operators and Conditions In Figure 4.12, we give an exemplary HTN involving Runtime-Worldstate-Modification-Nodes RWS that modify parameters we use in Operators OPs and Conditions CONs for realizing repeated executions in an Ensemble Program. Because the HTN in in Figure 4.12 does not include any Complex-

[7] Currently, we express Split-Node as PNs without any operator in our reference implementation. Because we only require an indicator for the respective action required during the execution of an Ensemble Program, this is sufficient for correctly representing the desired concurrency of partial plans in an Ensemble Program.

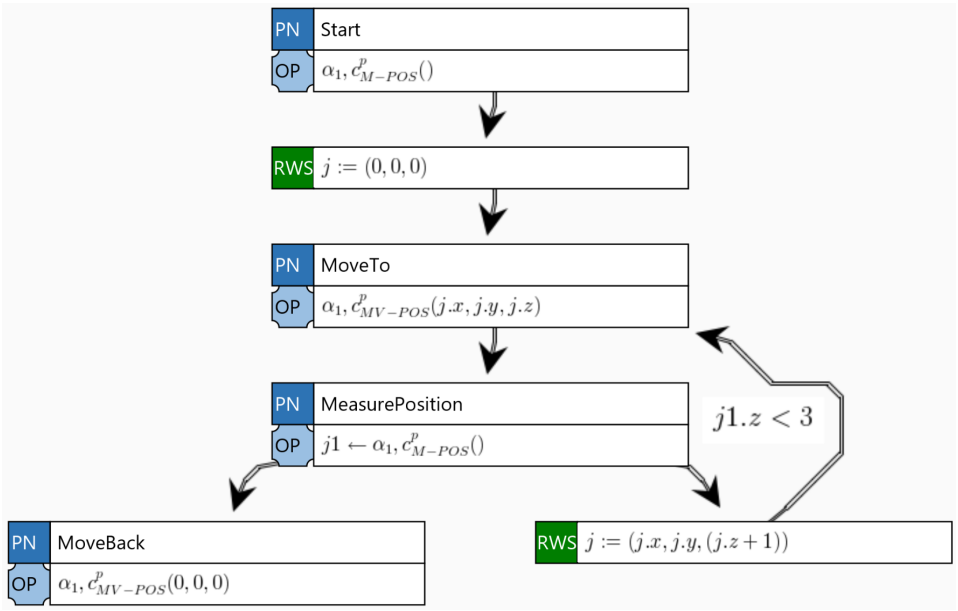


Figure 4.12: A MAPLE HTN involving Runtime-Worldstate-Modification-Nodes RWS that modify parameters we use in Operators OPs and Conditions CONs for realizing a repeated execution in the Ensemble Program. We do not depitct the resulting plan in addition as it has the same shape than the HTN due to the lack of Complex-Nodes. Figure exported from our prototypical reference implementation (cf. Section 3.3.5.1).

Node CN, there is no need for decomposition while running the planning algorithm. Instead, we focus on the Primitive-Node ⌈PN: *MeasurePosition*⌋, its default successor ⌈PN: *MoveBack*⌋ commanding a Identified-Planning-Agent $\alpha^\rho{}_1$ to move to the position $\langle 0,0,0\rangle$ by executing the capability $c^p_{\text{MV-POS}}$ in the instruction encoded in its only Operator OP, and its conditional successor, the Runtime-Worldstate-Modification-Node ⌈RWS: $\{j := (j.x, j.y, (j.z+1))\}$⌋. The plan resulting from the HTN in Figure 4.12 shows how the Ensemble Program's intended control flow. Because the first Runtime-Worldstate-Modification-Node ⌈RWS: $\{j := (0,0,0)\}$⌋ modifies the variable j of type 3D-COORDINATE, i.e., sets it to $\langle 0,0,0\rangle$, the succeeding instruction in the Operator ⌈OP: $\alpha^\rho{}_1$, $c^p_{MV\text{-}POS}$ $(j.x, j.y, j.z, 0, 0, 0)$⌋ listed in the Primitive-Node ⌈PN: *MoveTo*⌋ commands the Identified-Planning-Agent $\alpha^\rho{}_1$ to the position $\langle 0,0,0\rangle$ and thus the result $j1$ of letting $\alpha^\rho{}_1$ execute the capability $c^p_{\text{M-POS}}$ for measuring its current position in the Operator ⌈OP: $j1 \leftarrow \alpha^\rho{}_i, c^p_{M\text{-}POS}$⌋ again is this position $\langle 0,0,0\rangle$ in the first iteration of the Ensemble Program's execution. Thus, evaluating the Condition $j1.z < 3$ annotated to the conditional successor of ⌈PN: *MeasurePosition*⌋ results true. Consequently during the Ensemble Program's execution the variable j gets updated by executing the content of the Runtime-Worldstate-Modification-Node ⌈RWS: $\{j := (j.x, j.y, (j.z+1))\}$⌋ and the control flow is redirected back to the Primitive-Node ⌈PN: *MoveTo*⌋ whose parameter for instruction referenced in its only Operator OP then accesses the new value of j. As the ensemble programmer that designed the HTN in Figure 4.12 we thus expect for a correct execution of the Ensemble Program generated from the resulting plan, that one agent $\alpha \in \mathcal{E}^\rho$ first moves to the coordinate $\langle 0,0,0\rangle$, second ascends to the coordinate $\langle 0,0,1\rangle$, third ascends to the coordinate $\langle 0,0,2\rangle$, and fourth returns to the coordinate $\langle 0,0,0\rangle$.

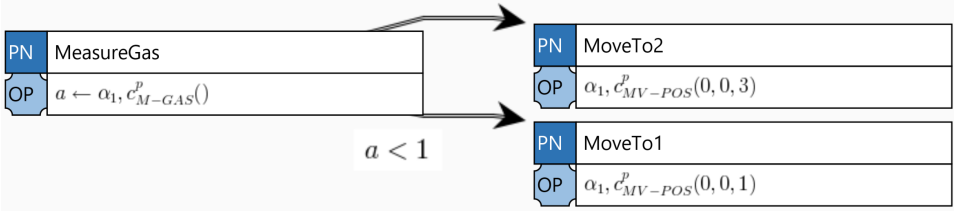


Figure 4.13: A HTN generating a IF/ELSE construct in the Ensemble Program. Figure exported from our prototypical reference implementation (cf. Section 3.3.5.1).

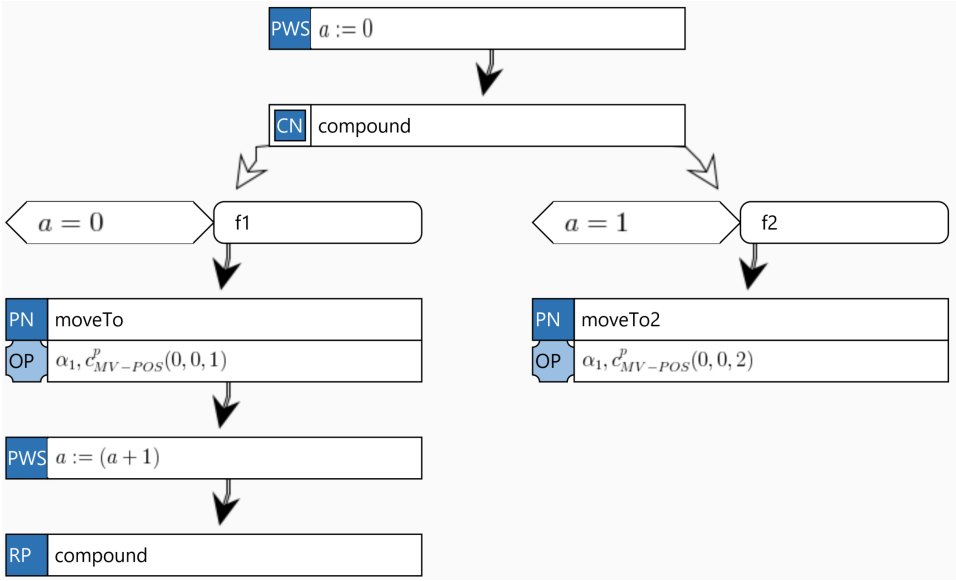
If/Else Constructs Using Temporary Variables In Figure 4.13, we see a plan expressing a conditional IF/ELSE construct using the possibility for directly accessing the result of executing an instruction in the same Ensemble Program without requiring to store it to Worldstate WS using an Runtime-Worldstate-Modification-Node RWS. We use this functionality for deciding on the respective succeeding node of an initial Primitive-Node ⌈*PN: MeasureGas*⌋ during the execution of an Ensemble Program. In the Operator ⌈*OP:* $a \leftarrow \alpha^{\rho}{}_1, c^p_{M\text{-}GASG}$⌋, we store the result of executing the capability $c^p_{\text{M-GASG}}$ in the temporary variable a. For deciding on the successor of this Primitive-Node ⌈*PN: MeasureGas*⌋, we can evaluate the value of a when executing the Ensemble Program and continue with the respective conditional successor ⌈*PN: MoveTo2*⌋ or with the default successor ⌈*PN: MoveTo1*⌋.

With the possibilities to express control structures for repeated and conditional program control flow, we propose in MAPLE, the ensemble programmer thus has all necessary functionality at hand to design complex Ensemble Programs containing one or multiple of them independently and possibly integrated.

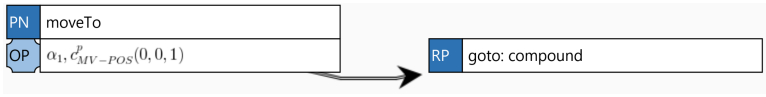
4.5.1.4 Replanning

We allow the user to express the need for new Ensemble Programs, i.e., for generating a new plan, when designing an HTN. Therefore, the user can include a Replanning-Node RP at the end of a partial plan. When executing the respective Ensemble Program, the executing ensemble then starts generating a new plan with a potentially modified Worldstate WS. In Figure 4.14, we depict an example for such. The design of the HTN in Figure 4.14a is intended to be very similar to that in Figure 4.10a enabling us to investigate in the different consequences for the respectively resulting ensemble plans.

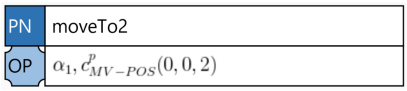
We make the following changes to the HTN from Figure 4.14a. We change the conditions in the methods for decomposing the Complex-Node ⌈*CN: compound*⌋ for the example in this section slightly avoiding to many redundant plans, i.e., decompose it using the Method ⌈*M: f1*⌋ if a is set to 0 in WS and with the Method ⌈*M: f2*⌋ if a is set to 1 in WS. Further, instead of letting the Primitive-Node ⌈*PN: MoveTo*⌋ point back to the Complex-Node ⌈*CN: compound*⌋ with a default successor, we point to a newly introduced Replanning-Node ⌈*RP: compound*⌋. We see the difference in decomposition when focusing on the respective plans in Figures 4.10b, 4.14b and 4.14c that get automatically generated from the HTN under respective conditions defined by WS. While the plan in Figure 4.10b includes the Primitive-Node ⌈*PN: MoveTo*⌋ multiple times using different parameters, the plan in Figure 4.14b derived from the same initial conditions concerning the variables defined in the Worldstate WS includes ⌈*PN: MoveTo*⌋ only one time before pointing to the aforementioned Replanning-Node ⌈*RP: compound*⌋. During the



(a) User-defined HTN.



(b) Resulting plan for initial conditions.



(c) Resulting plan with modified WS.

Figure 4.14: A MAPLE HTN involving Replanning-Nodes RP in Figure 4.14a and the resulting plans in Figures 4.14b and 4.14c. Figures exported from our prototypical reference implementation (cf. Section 3.3.5.1).

execution of the respective Ensemble Program, the ensemble programmer thus commands the executing ensemble to generate a new plan starting from the Complex-Node ⌈CN: *compound*⌋ encoded in the ⌈RP: *compound*⌋ with the updated WS. Because the variable a in WS is no longer set to 0 like commanded in the initial Planning-Time-Worldstate-Modification-Node ⌈PWS: $\{a := 0\}$⌋ but updated to 1 after the first planning by executing the statement encoded in teh Planning-Time-Worldstate-Modification-Node ⌈PWS: $\{a := (a + 1)\}$⌋, planning then results in the plan we depict in Figure 4.14c consisting of the Primitive-Node ⌈PN: *moveTo2*⌋ only. Compared to the plan in Figure 4.10b, we now can use different ensembles for executing the different Ensemble Programs generated by each execution of automated planning. This generates a new degree of flexibility the ensemble programmer can use if this is required for its use-case, i.e., for the switch in phases of SCORe missions (cf. Chapter 2).

4.5.2 Recapitulating the Example from the Problem Definition

We briefly recapitulate our example from Section 4.2.1 for demonstrating the benefits of our approach in comparison to the solution we used there. We thereby evaluate the expressiveness of MAPLE for defining such a mission concerning the provided control structures and Planning-Agent-Groups. We depict our solution in a HTN in Figure 4.15. Because we want to have the composition of the ensemble stay the same throughout the described mission, we do not include any Replanning-Nodes RPs in the HTN that would otherwise generate new Ensemble Programs

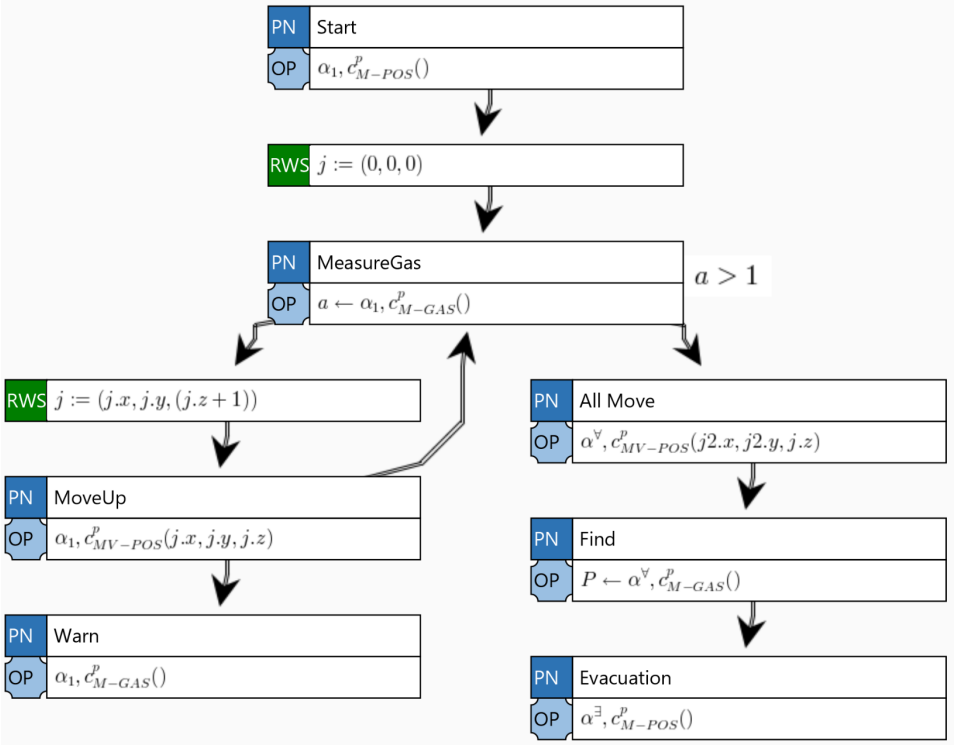


Figure 4.15: An exemplary solution for solving the example from Section 4.2.1 using the concepts of MAPLE. Figure exported from our prototypical reference implementation (cf. Section 3.3.5.1). Because our current implementation does not yet provide all capabilities required, we use respective equivalents, e.g., substitute WARN, EVAC, and $c^p_{\text{M-PERS}}$ with $c^p_{\text{M-GASG}}$. We an switch to the intended capabilities as soon as they are provided by simply adapting the existing saved HTN from our library.

at run-time we would generate new ensembles for (cf. Chapter 5). Further, because we did not include any alternative solutions for different situations we could express in the Worldstate WS that might be of relevance at planning time in the example from Section 4.2.1, the HTN in Figure 4.15 does also not contain any CNs. We thus describe the only partial plan including different Primitive-Nodes PNs, different Runtime-Worldstate-Modification-Nodes RWS, and conditional successor dependencies of those. We make use of different Planning-Agent-Groups for increasing the flexibility in extending the program for more agents.

In an initial Primitive-Nodes $\lceil$*PN: init*$\rfloor$, we define the required number of agents we want to involve in the final Ensemble Program by using the Set-Agent $\{\alpha^\rho_1, ..., \alpha^\rho_n\}$ in the listed Operator $\lceil$*OP:* $\{\alpha^\rho{}_1, \alpha^\rho{}_2, \alpha^\rho{}_3\}, c^p_{M\text{-}GASG}()\rfloor$, i.e., use three agents like in Section 4.2.1. To realize the repeated execution of the capability $c^p_{\text{M-GASG}}$ in increasing heights, we create a LOOP-construct using default and conditional successors for the Primitive-Nodes $\lceil$*PN: MeasureGas*$\rfloor$ and $\lceil$*PN: MoveUp*$\rfloor$, involving the variable j we initialize as a 3D-COORDINATE with the coordinate $\langle 0,0,0\rangle$ in the Worldstate WS with a Runtime-Worldstate-Modification-Node $\lceil$*RWS:* $\{j := (0,0,0)\}\rfloor$. In the LOOP-construct, we iteratively increase j in another $\lceil$*RWS:* $\{j := (j.x, j.y, (j.z+1))\}\rfloor$. We then use j for increasing the height of $\alpha^\rho{}_1$ when executing the instruction encoded in the Operator $\lceil$*OP:* $\alpha^\rho{}_1$, $c^p_{M\text{-}GASG}\rfloor$ listed in the Primitive-Node $\lceil$*PN:*

MeasureGas⌋ and for terminating the LOOP-construct, when reaching a higher height than 100 (cf. conditional successor annotated with $j.z > 100$ pointing from the Primitive-Node ⌈PN: *MoveUp*⌋ to the Primitive-Node ⌈PN: *Warn*⌋). We include a further termination criteria for the LOOP-construct using a conditional successor pointing from the Primitive-Node ⌈PN: *MeasureGas*⌋ to ⌈PN: *All Move*⌋ which we annotate with the condition $a > 1$, indicating that the execution of the capability $c^p_{\text{M-GASG}}$ of $\alpha^\rho{}_1$ commanded with the instruction in the Operator ⌈OP: $a \leftarrow \alpha^\rho{}_1, c^p_{M\text{-}GASG}$⌋, listed in ⌈PN: *MeasureGas*⌋, returned a critical gas concentration. In that case, we let all agents in the ensemble move upward to the respective required height using the Planning-Agent-Group $\alpha^\rho_\forall$ in the Primitive-Node ⌈PN: *All Move*⌋ and then execute their capability $c^p_{\text{M-PERS}}$ for detecting endangered persons. We then let any of the agents within the ensemble take up the temporary result P of the instruction's execution in the listed Operator ⌈OP: $P \leftarrow \alpha^\rho_\forall, c^p_{M\text{-}PERS}$⌋ to execute the evacuation of possibly endangered persons within ⌈PN: *Evacuation*⌋ using the Any-Agent $\alpha^\rho_\exists$ in the respectively enlisted Operator.

With this solution, we come by the drawbacks of the solution from Section 4.2.1.

1. Because in Multipotent System, we have no fixed configurations of agents and thus must not tailor programs generated with MAPLE for specific configurations. Instead, we define the requirements for an $\alpha \in \mathcal{A}_{\mathcal{MS}}$ that wants to fill the placeholders of Planning-Agents in a respective ensemble $\mathcal{E}^\rho$ by addressing capabilities to $\alpha^\rho \in \mathcal{A}^\rho$ in instructions.
2. Instead of explicitly encoding required coordination for executing the Ensemble Program in the individual programs of agents, we generate them implicitly from an autonomously generated plan derived from the HTN defining the mission. That way, the ensemble programmer can focus on the *what to execute* and let the *how to execute* up to the system, i.e., use the system's possibilities for self-organization.
3. By introducing a general pattern to exchange data within the ensemble using the shared WS and temporary variables, we avoid requiring to encode data exchange for every agent's program individually (cf. while we focus on the definition of Ensemble Programs in this chapter, we investigate in the details for executing them in Chapter 7).
4. Adding more agents to the Ensemble Program poses no more problem when using MAPLE. Increasing the size to 4, 5, or any other number of agents requires modifying only the very first Primitive-Node ⌈PN: *init*⌋ in our example (cf. Figure 4.15). Because communication within the ensemble is handled implicitly in the same way for additionally included agents than for already included ones and all other Primitive-Nodes in the HTN use appropriate Planning-Agent-Groups to address instructions, we do not require to make any further adaptations when increasing the number of agents.

4.5.3 Evaluating the Expressiveness of MAPLE in a Seeding Robot Scenario

To demonstrate our approach's generality for ensemble programming, we show how we can apply MAPLE to a completely different use case. In Figure 4.17, we give an exemplary HTN and the individual plans we derive when planning with it for a seeding mission in a farm-work scenario. We assume we have some field, where we want our Multipotent System system to sow seeds in a grid-style pattern (cf. Figure 4.16). The dimensions of the grid can vary and may have a different expansion in each of the two relevant dimensions. The Multipotent System supports the capabilities $c^p_{\text{MV-POS}}$, c^p_{SOW}, and $c^p_{\text{REFILLSEEDS}}$ in this scenario. By executing the capability $c^p_{\text{REFILLSEEDS}}$, an executing agent fills a random amount of seeds into a seeding

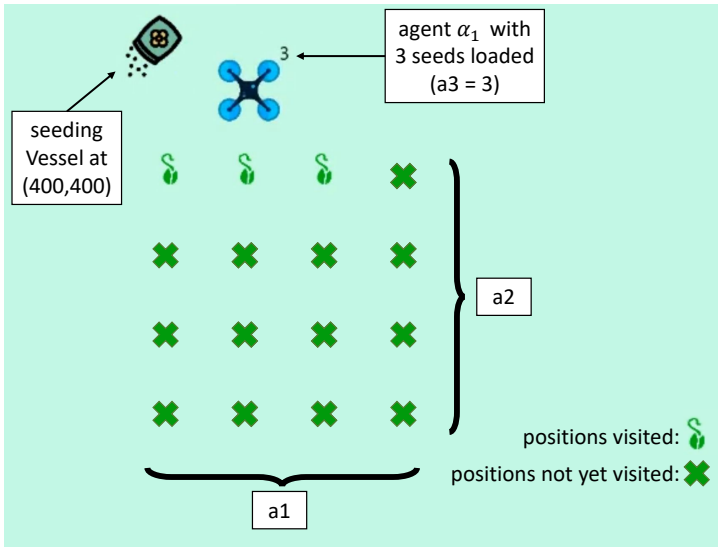


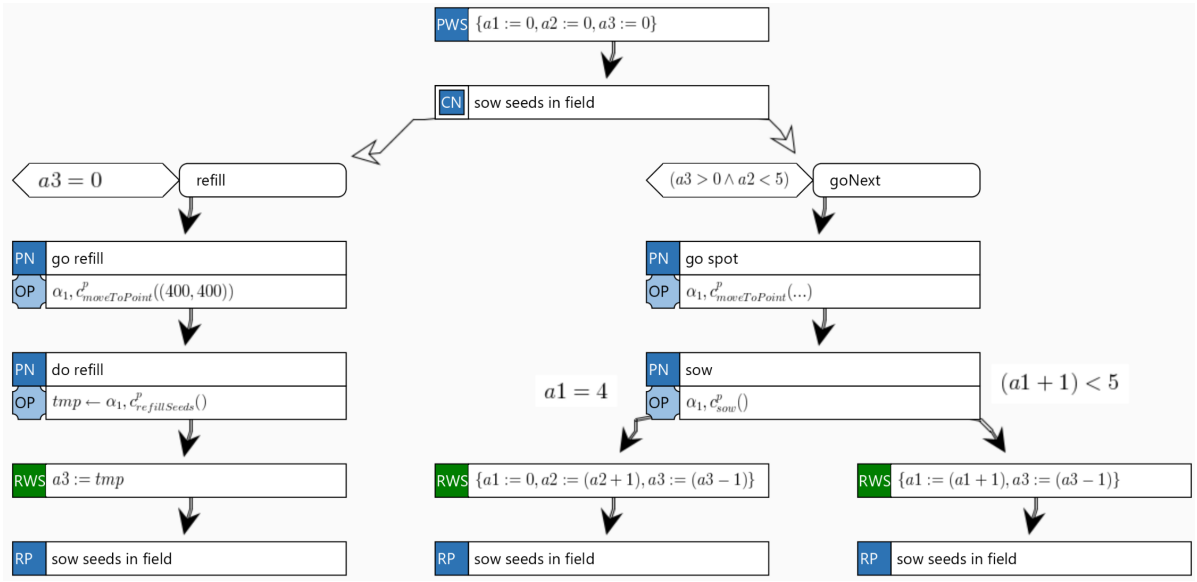
Figure 4.16: Enhanced screen-shot from a simple simulation environment environment depicting the situation in the seeding scenario (cf. video *MAPLE-Seeding-Robot* on GitHub and YouTube).

vessel. With executing c^p_{SOW}, an executing agent takes exactly one seed from the vessel and seeds it at its current position. Obviously, c^p_{SOW} should only be executed if there is at least one seed in the vessel.

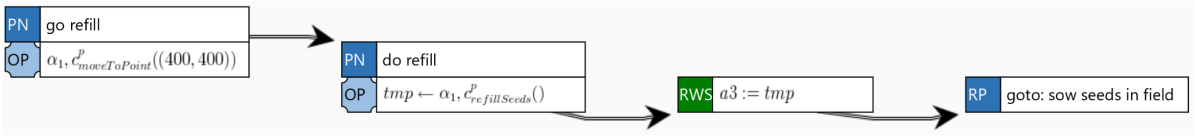
Using the concepts of MAPLE, an ensemble programmer can encode its requirements in an HTN easily. In an initial Planning-Time-Worldstate-Modification-Node PWS, we initialize the relevant variables in WS. The HTN in Figure 4.17a thus encodes the two dimensions of the grid where seeds are already deployed at in the variables $a1$ and $a2$ and offers the possibility to encode the current load of seeds in the vessel $a3$. Initially, we assume to have all these variables set to 0, i.e., no seed is deployed in the field yet, and there is no seed in the seeding vessel. Depending on the situation defined by WS, the Multipotent System should decide for the right plan.

If the seeding vessel is empty, i.e., $a3$ is set to 0, it should first be refilled with new seeds. Otherwise, seeds from the seeding vessel should be deployed to the field at a not already visited position. We can encode this in two different decompositions of the Complex-Node ⌈*CN: sow seeds in field*⌋ in the HTN in Figure 4.17a encoded in the respective Methods ⌈*M: refill*⌋ when $a3 = 0$ holds in WS and ⌈*M: goNext*⌋ when $a3 > 0$ and the field is not yet completely visited (i.e., $a2 < 5$ in case the field-describing grid expands for only 4 positions in one of its two dimensions).

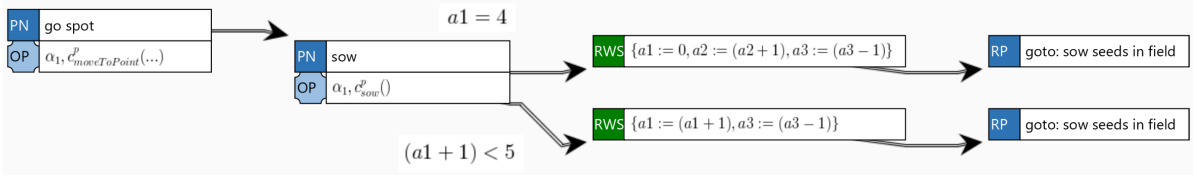
In case we decompose ⌈*CN: sow seeds in field*⌋ using ⌈*M: refill*⌋, we let one Identified-Planning-Agent $\alpha^\rho{}_1$ move the seeding vessel to the refilling spot (which we assume to be located at the 2-dimensional coordinate $\langle 400, 400\rangle$ in our example) by executing the instruction using the capability $c^p_{\text{MV-POS}}$ we encode in the Primitive-Node ⌈*PN: go refill*⌋ in a respetive partial plan we include in the HTN (cf. Figure 4.14b). Subsequently, $\alpha^\rho{}_1$ then should refill the seeding vessel by executing $c^p_{\text{REFILLSEEDS}}$ in the Primitive-Node ⌈*PN: do refill*⌋. Doing so produces the randomly set new number of seeds that are in the seeding vessel afterward, which we can store in the variable $a3$ in WS using a Runtime-Worldstate-Modification-Node ⌈*RWS:*



(a) User-defined HTN.



(b) Resulting plan for initial conditions.



(c) Resulting plan with modified WS.

Figure 4.17: An exemplary solution for accomplishing a seeding mission in a farmwork scenario using the concepts of MAPLE. Figures exported from our prototypical reference implementation (cf. Section 3.3.5.1).

$\{a3 := tmp\}\rfloor$ (where *tmp* indicates the result of executing $c^p_{\text{REFILLSEEDS}}$ in $\lceil$*PN: do refill*$\rfloor$). By letting a Replanning-Node $\lceil$*RP: sow seeds in field*$\rfloor$ finish this partial plan in the HTN, we let the autonomous planner generate the alternative plan using the Method $\lceil$*M: goNext*$\rfloor$ (except filling up the seeding vessel, unfortunately, resulted in 0 new seeds).

In that partial plan (cf. Figure 4.17c), we now can let one Identified-Planning-Agent $\alpha^\rho{}_1$ move to the next position in the grid we have not yet visited. We do this with an initial Primitive-Node $\lceil$*PN: go spot*$\rfloor$ by commanding the individual instruction including the capability $c^p_{\text{MV-POS}}$ in its respective Operator (we use some offsets for the exemplary coordinates there for improving the visualization in our reference implementation). After reaching the desired next spot, we let $\alpha^\rho{}_1$ execute the capability c^p_{SOW} for deploying a seed in the Operator enlisted in the Primitive-Node $\lceil$*PN: sow*$\rfloor$. To decide for the following position in the field to deploy a seed to, we then include an IF/ELSE-construct using conditional successors in the

partial plan. If the current row of the grid was not yet entirely visited by the preceding seeding action, we want to continue with the next spot in this row and thus update the variables in the Worldstate WS respectively using the Runtime-Worldstate-Modification-Node $\lceil$*RWS:* $\{a1 := (a1 + 1), a2 := a3 := (a3 - 1)\}\rfloor$. Doing so not only updates the position we want the executing ensemble to visit next but also reduces the number of seeds available in the seeding vessel. Thus, the replanning we command in a final Replanning-Node $\lceil$*RP: sow seeds in field*$\rfloor$ for that partial plan might result in a plan for refilling the seeding vessel first (as we described it previously) before continuing with seeding the next seed. If the current row of the grid instead is fully visited after executing the instruction enlisted in the Primitive-Node $\lceil$*PN: sow*$\rfloor$, we want the Multipotent System to continue sowing within the next row in the field in case we are not yet finished with the entire field. We encode this in a conditional successor for the Primitive-Node $\lceil$*PN: sow*$\rfloor$ that we take if the variable $a1$ in WS is still lower than the maximum number of rows (4 in our exemplary HTN in Figure 4.17). Then, we want to update the variables in WS accordingly in a Runtime-Worldstate-Modification-Node $\lceil$*RWS:* $\{a1 := 0, a2 := (a2 + 1), a3 := (a3 - 1)\}\rfloor$ before finishing the partial plan with a Replanning-Node $\lceil$*RP: sow seeds in field*$\rfloor$ that again causes the generation of a new plan.

With the HTN and the individual plans for the different conditions described by variables in WS we demonstrate in Figure 4.17, we thus showcased the expressiveness of our approach in another scenario we initially did not have in mind while designing the approach. Thus, this gives another example proving the potential expressiveness of MAPLE. We provide video materials showing an exemplary run of the seeding robot scenario on GitHub and YouTube[8] (video *MAPLE-Seeding-Robot*).

4.5.4 Evaluating the Expressiveness of MAPLE within a Forest Fire Scenario

We further evaluate the expressiveness of our approach by applying it to the use case of *Dealing with Forest Fires* (cf. Section 2.6). We thereby use the functionalities we deliver in MAPLE that can be beneficial for solving the problem defined in the case study. Therefore, we design a HTN consisting of different partial plans PART-$\rho_1, \ldots,$ PART-ρ_4 (cf. Figure 4.17a) and present the different plans ρ_1 and ρ_2 resulting in the different possible situations defined by the variables in the Worldstate WS (cf. Figures 4.19b and 4.19c). To increase readability, we use the following abbreviations for the nodes occurring in Figure 4.19:

$\text{PN}_1 := \lceil$PN: *init*$\rfloor$ $\quad$ $\text{PWS}_1 := \lceil$PWS : $\{f_{ws} := Nil, F := \{\}, A := (0, 0, 40, 40)\}\rfloor$

$\text{PN}_2 := \lceil$PN: *observe area*$\rfloor$ $\quad$ $\text{PWS}_{4-A} := \lceil$PWS : $F \leftarrow F \cup \{f_{ws}\}\rfloor$

$\text{PN}_3 := \lceil$PN: *handle fire*$\rfloor$ $\quad$ $\text{PWS}_{4-B} := \lceil$PWS : $f_{ws} := Nil\rfloor$

$\text{CN}_1 := \lceil$CN : *init*$\rfloor$ $\quad$ $\text{RWS}_2 := \lceil$RWS:$f_{ws} := f\rfloor$

$\text{RP}_2 := \lceil$*RP: keep area fire-save*$\rfloor$

In a first partial plan $\rho_1^{\text{PART}} := [PWS_1, PN_1]$ we include in the HTN in Figure 4.17a, we initialize the relevant variables in the Worldstate WS. the Planning-Time-Worldstate-Modification-Node PWS_1 thus initializes the variables f_{WS} with Nil, initializes the area of interest A with $\langle 0, 0, 40, 40\rangle$ (that combines the x and y coordinates of a 3D-COORDINATE with

[8] https://github.com/isse-augsburg/ensemble-programming or https://github.com/kosakoliver/ensemble-programming or https://www.youtube.com/user/ISSELabs

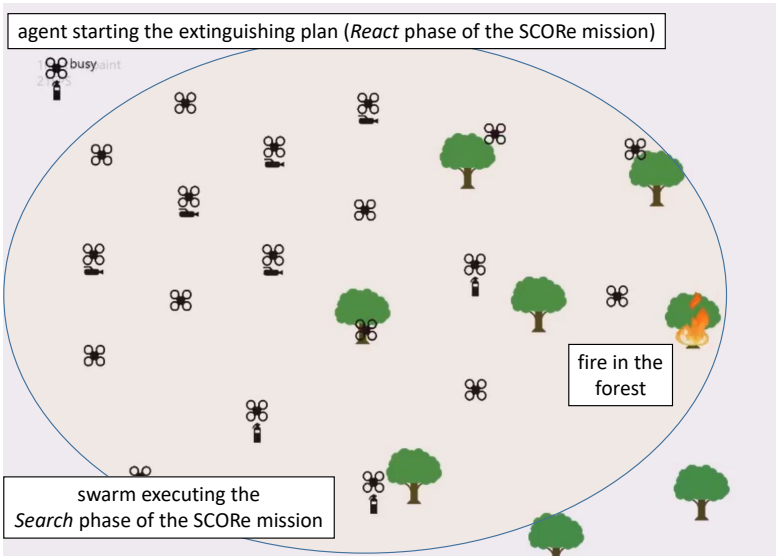


Figure 4.18: Enhanced screen-shot from a simple simulation environment environment depicting the situation in the forest fire scenario shortly before a fire is detected (cf. video *MAPLE-Forest-Fire-Planning-Execution* on GitHub and YouTube).

a DOUBLE for length and a DOUBLE for width) and a set of 3D-COORDINATES F to the empty set $\{\}$. The Primitive-Node PN_1 then directs the whole ensemble to the center of the forest at $x = 20$ and $y = 20$, which we want to survey at an altitude of $z = 50$ meters. We achieve this by using an instruction addressing the capability $c^p_{\text{MV-POS}}$ to the postition $\langle 20, 20, 50\rangle$ to the Planning-Agent-Group $\alpha^\rho_\forall$ which we include in the Operator $\lceil$*OP:* $\alpha^\rho_\forall$, $c^p_{MV\text{-}POS}$ *(20,20,50)*$\rfloor$ enlisted in the PN 1.

For the situation that we do not know any location of a fire in the forest, i.e., the variable p_{fire} in the Worldstate WS has the value Nil, we design the partial plan PART-ρ_2 := [PN_2, RWS_2, RP_2]. Therein, we let a swarm of agents $\alpha^\rho_{\{10\}}$ consisting of a minimum of 10 and a maximum of 50 agents execute a Collective Capability to equally distribute in the area of interest (A) using the potential field algorithm encapsulated in a virtual capability c^v_{POT} in PN_2. We assume that the swarm can autonomously adapt the altitude for gaining surveillance quality according to the number of swarm members like it is proposed to be achievable by Villa et al. [2016a]. We can achieve such behavior with an appropriate implementation of the respective Collective Capability c^v_{POT} (we explain how we can achieve this in Chapter 7). In that partial plan, we use the capability c^p_{DNF} for detecting new fires (i.e., such not already included in F) on the ground as the parameter of c^v_{POT}. That way, the Collective Capability c^v_{POT} returns the position of a fire f as a 3D-COORDINATE derived from the detecting agent's position with $f := \langle f_x, f_y, 0\rangle$ as soon as one member of the swarm executing it detects a fire (cf. Chapter 7 for the execution behavior andtermination of Collective Capabilities). Detecting a fire then causes an update of the world state in the Runtime-Worldstate-Modification-Node RWS_2 that sets the variable f_{ws} in to the result of c^v_{POT}, i.e., $f_{ws} := f$, followed by a Replanning-Node RP_2 referencing the only Complex-Node CN in the HTN (keep area fire-save).

For the situation that the ensemble is aware of a fire, i.e., the world state holds a respective entry $f_{ws} \neq Nil$ so that we use the method $\lceil$*M: observe and clear area*$\rfloor$ to decompose $\lceil$*CN:*

keep area fire-save] (cf. Figure 4.19a), we design two concurrent partial plans PART-$\rho_3 := [\text{PN}_3]$ and PART-$\rho_4 := [\text{PWS}_{4\text{-A}}, \text{PWS}_{4\text{-B}}, \text{PN}_2, \text{RWS}_1, \text{RP}_2]$ we want the ensemble to execute in that situation. In ρ_3^{PART}, we instruct α_1^ρ to execute the capability c^p_{EXT} for extinguishing the fire at the identified location f_{ws} in a respective instruction. Further, we instruct the Any-Agent $\alpha^\rho_\exists$ to stream a video from the position f_{ws} towards the user by executing a respective capability c^p_{STR}. We include both instructions in a respective Operator in PN_3. In PART-ρ_3, we can thus let the system decide with respect to the current configuration of agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ whether one agent is sufficient for executing that plan (i.e., $\alpha_1^\rho = \alpha^\exists$) or two agents are required instead (i.e., $\alpha_1^\rho \neq \alpha^\exists$) for executing the respective Ensemble Program. As parameter for both, c^p_{STR} and c^p_{EXT}, the planning process generates a copy of the concrete position of the fire, e.g., if $f_{ws} := \langle 23, 47, 11\rangle$, we use the parameter $\text{PAR}_{c^p_{\text{EXT}}} := \langle 23, 47, 11\rangle$ and $\text{PAR}_{c^p_{\text{STR}}} := \langle 23, 47, 11\rangle$. We need that copy because in an the concurrent partial plan ρ_4^{PART}, we add the identified location of the fire f_{ws} to a set of known fires F in $\text{PWS}_{4\text{-A}}$ and then reset f_{ws} to Nil in $\text{PWS}_{4\text{-B}}$ before an other ensemble again executes *observe area* in the subsequent partial plan PART-ρ_2.

Planning with the definitions in the HTN from Figure 4.19 then results in a plan consisting of PART-ρ_1 combined with PART-ρ_2 if $f_{\text{WS}} = Nil$ holds in the world state WS (cf. Figure 4.19c). If otherwise $f_{\text{WS}} \neq Nil$, i.e., a fire was detected by an ensemble previously, a plan consisting of PART-ρ_3 concurrent to PART-ρ_4 results from that planning (cf. Figure 4.19b). We provide video materials showing the progress of building the HTN from Figure 4.19 and an exemplary execution of the respective plans in the firefighter scenario using a simple simulation environment (cf. Figure 4.18) on GitHub and YouTube[9] (videos *MAPLE-Forest-Fire-HTN-Design* and *MAPLE-Forest-Fire-Planning-Execution*).

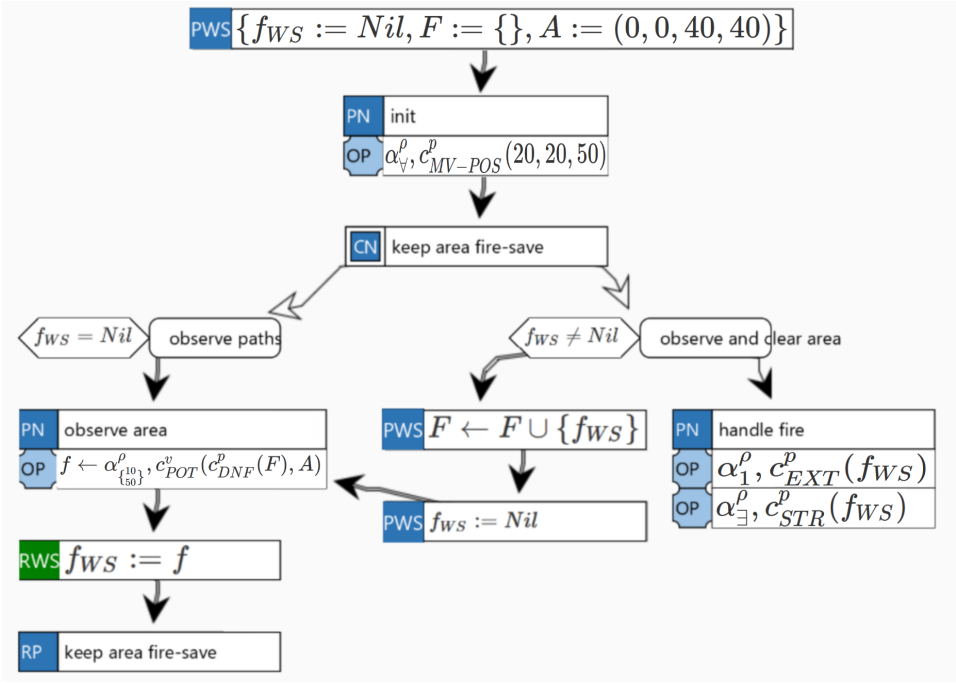
4.5.5 Comparing the Expressiveness of MAPLE to Other Approaches

To evaluate our approach and highlight its benefits for ensemble programming and multi-robot task orchestration, we investigate other concepts presented in the literature for their intercomparison. We depict relevant properties in Table 4.9 to categorize the different approaches on ensemble programming and task orchestration for MAS/MRS. After performing this classification we conclude on the strength and weaknesses of the approaches when comparing their expressiveness.

4.5.5.1 Task Orchestration with Dolphin

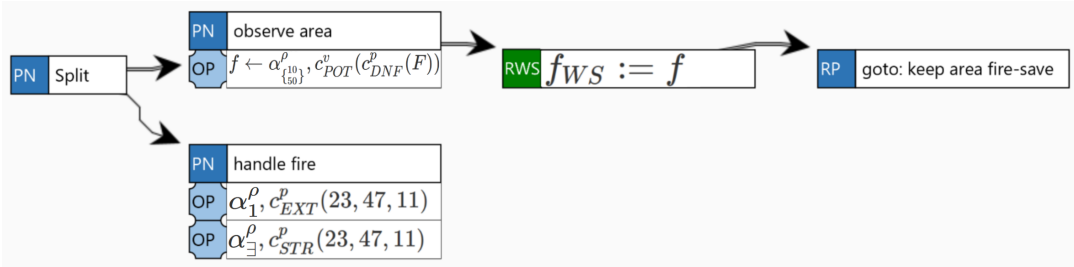
A framework providing a scripting language for multi-vehicle networks is Dolphin [Lima et al., 2018]. Like our approach, Dolphin supports instructing other types of robots than UAV in general, i.e., provides appropriate measures for instructing heterogeneous teams of mobile robots. A user can specify the task control flow in a Dolphin program to be sequential, concurrent, and event-based, i.e., include decisions in a step of a task that depends on previous steps in that task like. New Dolphin programs can be composed of existing ones, making the approach compositional. Events can also be time-dependent in Dolphin, allowing a user to define waiting for tasks and cancel the execution of certain tasks after a defined waiting time. There is no option in Dolphin allowing for parallel executions for multiple robots simultaneously. With Dolphin, a human operator can define tasks for particular robots and teams of robots without explicit knowledge of the concrete implementation of these tasks' execution. Dolphin provides

[9] https://github.com/isse-augsburg/ensemble-programming or https://github.com/kosakoliver/ensemble-programming or https://www.youtube.com/user/ISSELabs



(a) User-defined HTN.

(b) Resulting plan if initially no fire location is known (i.e., $f_{ws} = Nil$), concatenating ρ_1^{PART} and ρ_2^{PART}.



(c) Resulting plan if a fire location is known (i.e., $f_{ws} = Nil$), consisting of the concurrent partial plans ρ_3^{PART} and ρ_4^{PART}.

Figure 4.19: An example HTN consisting of situation-aware partial plans for handling the firefighter scenario from Section 2.6 including possible plans resulting from automated planning in Figure 4.19b and Figure 4.19b. Figures exported from our prototypical reference implementation (cf. Section 3.3.5.1).

Table 4.9: Comparison of expressiveness in ensemble programming concerning different aspects.

	Maple	Dolphin	PaROS	Voltron	Gutmann	Protelis	Meld	Buzz	TeCola
programming									
graphic	✓	–	–	–	–	–	–	–	–
imperative	–	✓	✓	✓	✓	–	–	✓	✓
functional	–	–	–	–	–	✓	–	–	–
logic	–	–	–	–	–	–	✓	–	–
task control structures									
sequential	✓	✓	–	✓	✓	–	–	–	–
parallel(p/l)	✓/✓	✓/–	–/–	(–/–)	✓/–	–/–	–/–	–/–	–/–
conditional	✓	✓	–	✓	✓	–	–	–	–
repeated	✓	✓	–	✓	✓	–	–	–	✓
concurrent	✓	✓	–	✓	✓	–	–	–	–
time-dependent	–	✓	–	✓	–	–	–	–	–
(intended)/ actual scale									
very high	(✓)/✓	–	–	–	?	(✓)/✓	(✓)/✓	(✓)	–
high	–	–	(✓)	–	?	–	–	(✓)/✓	(✓)
low	–	(✓)/✓	✓	(✓)/✓	?	–	–	–	✓
very low	–	–	–	–	?	–	–	–	–
program generation									
run-time	✓	–	–	–	–	–	–	–	✓
design-time	–	✓	✓	✓	✓	✓	✓	✓	✓
compositional	✓	✓	✓	–	✓	(✓)	–	✓	✓
control level									
individuals	✓	✓	✓	–	✓	–	–	–	✓
ensembles/teams	✓	–	–	✓	✓	–	–	–	✓
aggregates/swarms	✓	–	✓	–	–	✓	✓	✓	–
group support									
All-Agent	✓	–	✓	✓	–	✓	✓	✓	✓
Any-Agent	✓	–	–	✓	–	–	–	–	–
Swarm-Agent	✓	–	✓	–	–	✓	✓	✓	–
Set-Agent	✓	–	–	–	✓	–	–	–	–
system type									
homogeneous	✓	✓	✓	✓	–	✓	✓	✓	–
heterogeneous	✓	✓	–	–	✓	–	–	(✓)	✓
devices									
mobile robots	✓	✓	✓	✓	✓	(✓)	(✓)	✓	✓
sensor networks	–	–	–	–	–	✓	–	–	–
examples									
SAR	✓	–	–	✓	✓	✓	–	–	–
Surveillance	✓	✓	✓	✓	–	✓	–	–	✓
Env.-Monitoring	✓	–	–	✓	–	–	–	–	–
Major Catastrophe	✓	–	–	–	–	–	–	–	–
deployed to									
real-world	(✓)	✓	✓	✓	–	✓	–	–	–
simulation	✓	–	✓	✓	–	✓	✓	✓	✓
expertise									
domain	✓	✓	✓	✓	✓	✓	✓	✓	✓
coding	–	✓	✓	✓	✓	✓	✓	✓	✓

some useful abstractions from concrete task allocation at program design time, letting the system executing a mission decide on the concrete instance at run-time. This allows the designer to specify some parts in the mission to be executed in parallel or concurrent with operators requiring "oneOf" or "allOf" the tasks to be executed by the system [Lima et al., 2018]. There is no support for Planning-Agent-Groups like we provide in Maple for the All-Agent, the Any-Agent, the Swarm-Agent, and the Set-Agent. Unfortunately, in general, tasks need to be programmed beforehand and can not be introduced to a system that is already running, which we inherently support with our planning-based approach. While a user can define tasks for pre-formed robot teams with Dolphin, it does not support abstract programming of groups for addressing all, any, or a certain set of robots in a team to execute a specific action. In the examples provided by Lima et al. [2018], at most, four robots are involved in a mission, i.e., the approach currently is designed for very low-sized systems. Further, Dolphin offers no possibility for exploiting emergent effects of collective behavior like swarm behavior can deliver

and instead postpones such support to future work. Further, Dolphin does not include the possibility for online and situation-aware replanning that can generate new tasks at run-time, which is a beneficial option we can exploit in MAPLE by using automated planning with HTN if this is of use. In Dolphin, programs need to be written in a code-heavy style requiring the domain and technical expertise for designing correct Dolphin programs. MAPLE instead provides a graphical user front-end, potentially opening its use to a broader range of users that are not familiar with code generation. Examples provided for Dolphin in literature are restricted to surveillance problems that are realized with a real-world system.

4.5.5.2 Task orchestration with TeCola

With the domain-specific language, TeCola [Koutsoubelias and Lalis, 2016] dedicated to programming missions for heterogeneous teams of robots, users can instruct MAS/MRS on an abstract level using the Python programming language. By abstracting the robots and capabilities of robots as services available to the user while programming, TeCola reduces the complexity of the internal implementation of these services. TeCoLa supports creating teams of such robots represented as a service that a user can address in its programs as a whole efficiently. Teams then can be instructed by a programmer as a whole, i.e., all agents in a team need to execute the individual instruction. Thus TeCola supports an equivalent for the All-Agent we support in MAPLE. Unfortunately, there is no possibility to specify that any agent, a set of agents, or a swarm of agents from a predefined team should execute a specific instruction like it is possible with Planning-Agent-Groups in MAPLE. While TeCola also eases the programming with primitives to instruct such teams within missions, TeCola still requires fine-grained management of robots involved in those missions during task specification. TeCola does not support self-organized collective behavior like we do, e.g., by integrating swarm algorithms, but instead relies on explicit coordination in robot teams under all circumstances. Instead of a decentralized approach for controlling robot teams, TeCoLa uses a master-slave architecture, producing a single-point-of-failure for all team-level operations. In the sense of TeCoLa uses the term in the descriptions of their framework, a swarm is only used as an equivalent used if "several nodes need to perform the same operations" [Koutsoubelias and Lalis, 2016]. Thus, there is no possibility of exploiting emergent effects integrated into TeCoLa as we provide them in our approach. TeCoLa provides some type of situation-aware task generation and task introduction like we support them in our approach with plans derived from automated planning while the system is already running in a productive state. While all TeCoLa programs need to be finalized before the user starts the system, some predefined tasks can be newly introduced concurrently at run-time (i.e., event-based triggering of concurrent tasks). Currently, TeCola was only deployed in simulation in a Distributed Surveillance scenario. Thus, compared to MAPLE, TeCola shows deficits in expressiveness during program definition (no Planning-Agent-Groups) and deficits in flexibility during execution (no SO-support).

4.5.5.3 Task Orchestration with Gutmann

Another recent approach for task orchestration in MAS/MRS is that of Gutmann and Rinner [2021] (called Gutmann for short in the following as there is no other acronym given in the literature). Programming with Gutmann aims at specifying missions for multi-UAV applications in an easy-to-read fashion. The approach provides measures for specifying different tasks within a mission that can be orchestrated to be executed sequentially, in parallel, conditionally,

repeated, and concurrent. Like we do in MAPLE, instructions in Gutmann are defined on the capability level of robots. Currently, [Gutmann and Rinner, 2021] specifically restricts the usage to UAV exclusively. In Gutmann programs, these UAV need to be addressed individually. Instead of leaving more flexibility to the executing system for allocating tasks to agents like we aim at in MAPLE with the All-Agent, the Any-Agent, and the Swarm-Agent, the programmer needs to perform a full task specification design-time. By introducing teams of drones that can be instructed abstractly, Gutmann programs offer support for the easier instruction of that teams, i.e., provide an equivalent to the Set-Agent we offer in MAPLE. Up to now, Gutmann programs can not involve swarm behavior in the sense we support it with Collective Capabilities in MAPLE. Further, there is no measure for online task generation, i.e., autonomous reaction to situation changes with replanning or replanning-like measures. Gutmann has not yet been deployed to any simulated nor real-world system. Thus, the practicability of the Gutmann approach is still post-phoned to future work. While they propose their approach should be applicable domain independently, in their running example Gutmann and Rinner [2021] focus on a SAR mission solely and do not provide other examples — which might follow in their future work as [Gutmann and Rinner, 2021] is a position paper. This also makes it hard to guess the intended size of the goal system. Gutmann promises to be a suitable option for task orchestration in multi-robot applications for flying ensembles in the future. Nevertheless, it has some drawbacks, like the need for centralized coordination from a ground control station that takes responsibility for all system coordination except the direct hardware access which is performed onboard. Further, the lack of easy access to swarm behavior in Gutmann combined with the need to precisely predefine tasks according to the individual drones' hardware configuration must be equalized before competing with the expressiveness of MAPLE.

4.5.5.4 Task Orchestration with Voltron

Another approach providing a framework for task orchestration aiming at MAS/MRS is Voltron [Mottola et al., 2014]. With their approach, the authors propose an appropriate abstraction from particular robots towards programming on the team level. With this abstraction and for the applications they aim at, the authors hide complexity arsing in coordinating teams of robots and provide scaling and concurrent task execution measures from the programmer. For achieving this, on the one hand, Mottola et al. [2014] abstract the whole system into one abstract device the user can write programs for. Thus, the programmer in general programs for a single "abstract drone" [Mottola et al., 2014], i.e., writes a program if it was not intended for a team/ensemble of robots but only for one. That way, Voltron supports some kind of the All-Agent and Any-Agent like we do in MAPLE: Depending on the concurrent tasks included in a Voltron program, the underlying system can choose how many robots concurrently work on the program. On the other hand, they restrict the user's possibilities for programming that device by an API restricted to a set of pre-coded algorithms. Up to now, with Voltron, a user can not specify collective behavior in the form of swarm algorithms, i.e., there are no team-level implementations adopting swarm behavior and exploiting the advantageous emergent effects that possibly can arise from them for an application. Voltron programs further need to be fully specified at design-time. There is no possibility for introducing new tasks to the executing system at run-time. Also, the Voltron API does not support the composition of complex programs from simpler ones (no modularity is provided in Voltron). While Voltron does include a mechanism to compensate for failures at run-time, e.g., to maintain the execution of once-defined tasks, it does not support other situation-aware modifications of missions.

Further, Voltron relies on either a centralized execution or a strictly synchronized execution which can quickly produce a single-point-of-failure in the first case and can reduce execution speed in the latter case. Further, there is no possibility for an autonomous generation of tasks at run-time in Voltron like we can provide it with MAPLE. By exclusively focusing on providing a possibility for programming on the team level, Mottola et al. [2014] further loses the ability for controlling and specifying tasks for individual robots. All robots in teams are supposed to execute the same actions in a mission by the authors, making it hard to create complex missions as we aim at in the context of SCORe missions. Moreover, all robots programmed with Voltron need to be designed homogeneously, i.e., provide the same capabilities, reducing the range of possible applications. Current applications of Voltron involve up to 7 robots in a real-world Distributed Surveillance task where a set of points of interest need to be analyzed by one robot each. Other stimulative experiments mentioned by Mottola et al. [2014] that are not further investigated in their descriptions involve Environmental Monitoring and a SAR scenario. Thus, compared to MAPLE, Voltron especially lacks flexibility in task design by only involving heterogeneously configured agents only and neglects exploiting homogeneity, e.g., by integrating swarm behavior.

4.5.5.5 Ensemble Programming with Meld

Meld, an approach for abstract ensemble programming from Ashley-Rollman et al. [2009], provides other benefits to programmers. With its logic programming approach, Meld can generate complex programs from a minimal fact set which needs to be defined by the programmer. While these base facts need to be defined by the programmer, Meld can deduce programs from them autonomously. On the downside, programs automatically generated that way are hard to comprehend and retrace by a user. Further, there is no existing solution for reusing partial programs in a modular fashion, i.e., composing new programs from already existing ones is not possible with Meld. Each program repeatedly needs to be deduced from the base facts, consuming a lot of computational resources. Up to now, Meld also lacks a demonstration of real-world usage. Instead, Meld focuses on other types of ensembles than that we want to control with our approach. In Meld, an ensemble consists of millions of independent robot-like entities (the authors assume to have miniaturized robots with minimal locomotion capabilities). Those are used for abstractly modeled, simulated, and self-shaping large-scale ensembles. Because programming with Meld always addresses all homogeneous entities in the ensemble to execute a particular operation, we can say that there is some support for programming all entities or a swarm of entities like we do with the All-Agent and the Swarm-Agent in MAPLE. Besides that, we can not find equivalents to the Any-Agent or the Set-Agent in Meld, nor can an ensemble programmer address individual agents in an isolated fashion. With its completely different focus on ensemble programming, Meld is no option for commanding mobile robots in the real world, e.g., within SAR scenarios or Environmental Monitoring..

4.5.5.6 Aggregate Programming with Protelis

A further approach focusing on the abstract programming of aggregates is Protelis [Pianini et al., 2015]. Protelis provides some useful primitives that abstract actions of whole aggregates a user can integrate into programs. Thereby, Protelis also delivers guarantees concerning the so-created programs like the self-stabilization of calculations performed with that aggregate primitives. While these properties would be favorable for ensemble programming in general,

Protelis, unfortunately, lacks the possibility of task orchestration. With Protelis, a user can only instruct an aggregate of devices to execute one task at a time. There is no possibility to define sequential, parallel, concurrent, or alternative behavior outside of individual Protelis programs. Thus, in Protelis, a task concept is missing in general (of course, within Protelis programs, the aforementioned control structures can be used). This fact makes it challenging to construct complex solutions for missions situated in the real-world that require event-based reactions to changed conditions. Protelis provides some measures for composing new programs from existing modules, i.e., from aggregate primitives. Besides this, it is complicated to construct new Protelis programs from existing ones for creating more complex ones integrating the results of each other as there is no possibility for task-orchestration included in Protelis (thus we mark the respective entry *compositional* in Table 4.9 only partially). Consequently, Protelis does not support the run-time generation of new programs for the aggregate, i.e., Protelis programs need to be defined entirely at design-time. Because Protelis is intended to work with whole aggregates consisting of massive numbers of participating devices, there is implicit support for commanding all agents, i.e., an equivalent to the All-Agent in Maple. Additionally, Protelis provides some support for pre-selecting sub-aggregates for executing specific programs, i.e., Protelis supports some equivalent to the Swarm-Agent in Maple. Besides this, there is no support for the Any-Agent or the Set-Agent, nor for controlling individual agents as we do in Maple. Using the paradigm of spatial computing and primarily focusing on homogeneous devices (Protelis supports some measures for filtering devices according to their properties, potentially allowing for deploying it to heterogeneous systems), the evaluation of aggregated results is easy to do with Protelis. Its usage for commanding robots, especially for heterogeneous and mobile ones, has not yet been demonstrated, which we think is justified by the different understanding Protelis has when speaking of an ensemble. Instead of programming mobile robots, Protelis is more suitable for programming senor networks where data aggregation, distribution, and information processing with massive numbers of spatially distributed devices is the focus. Unless the authors do not explicitly aim at controlling mobile robots using Protelis, we demonstrate how we could achieve this in principle in Section 7.6.2.5 by integrating and hosting a Protelis execution environment in Maple.

4.5.5.7 Swarm Programming with Buzz

Another programming language aiming at collective systems is Buzz [Pinciroli and Beltrame, 2016]. In comparison to Protelis, the authors of Buzz directly aim at integrating their programming language within robot operating systems. Buzz provides support for homogeneous and heterogeneous systems in general. This is achieved by an integrated selection mechanism used to separate devices into teams that are again homogeneously configured. For such homogeneously configured teams, Buzz provides swarm primitives for achieving a specific desired collective behavior each. There is no possibility for defining tasks involving heterogeneously configured devices working together. Unfortunately, Buzz also lacks a concept for goal-oriented task orchestration. While within a Buzz program a programmer can use sequential, conditional, and repeated control flow for solving a specific task, this is not possible on the level of tasks themselves. Currently, Buzz programs are deployed in simulation only. Caused by their lack in providing proper measures for task-orchestration, Buzz programs need to be fully defined at design-time and deployed to the respective devices. With Maple applied in Multipotent System, we can autonomously let the systems generate new situation-aware programs at run-time. Like for other approaches to swarm/aggregate programming, we can find an equivalent for the

All-Agent and the Swarm-Agent like we support it in Maple. Fortunately, Buzz programs can be composed of modules defined beforehand and do not require a complete generation from the scratch like e.g., Meld [Ashley-Rollman et al., 2009] requires them. When commanding a swarm in Buzz, this implicitly addresses all devices in that swarm, i.e., Buzz offers support for an All-Agent-like concept. Because the programmer can modify the size of that swarm, we can realize a Swarm-Agent-like task allocation with the concepts provided in Buzz. Applications that Pinciroli and Beltrame [2016] describe only involve movement patterns for swarms demonstrated in simulation. Formation flights depicted there involve high numbers of agents (up to 20). Further theoretical evaluations the authors provide, also scale up to very high numbers of participating devices (up to 1000). There is no current application of Buzz for a real-world scenario. In comparison to Maple, especially the lack in concepts for proper task orchestration and low support for heterogeneity makes Buzz impractical to use in many situations like they can occur in SCORe missions.

4.5.5.8 Ensemble Programming with PaROS

PaROS [Dedousis and Kalogeraki, 2018] is another approach for ensemble programming. It introduces primitives for collectives the user can define simple tasks with and let those tasks distribute within a swarm of UAVs. Possibilities in designing tasks currently are very restricted and do only involve path planning for area covering missions. There is no direct support for Planning-Agent-Groups like we provide in Maple, but because PaROS can only handle complete swarms, it indirectly supports a concept similar to the All-Agent, and the Swarm-Agent. Thus, PaROS lacks a concept for the Any-Agent and the Set-Agent and obviously also for addressing individual agents independently. Only homogeneously equipped UAVs are in the focus of PaROS, and there is no support for multi-robot systems in general. While PaROS supports some good abstractions for encapsulating particular swarm behavior in tasks for groups of UAVs, it does not aim to interconnect those tasks in complex programs with any parallel, concurrent, alternating, or iterated execution of different swarm algorithms we aim for. Experiments Dedousis and Kalogeraki [2018] performed with PaROS include up to 7 simulated or 4 real robots. Nevertheless, the authors claim that their approach scales with increased system sizes involving more robots. Thus, PaROS focuses on very small-sized systems but potentially can also work with highly scaled systems. In PaROS there is no possibility for the processing of once achieved results from executing a collective behavior. This is also the case because PaROS focuses on very similar searching tasks only that need to be programmed beforehand. This means, only parameters of tasks and algorithms used for that restricted set of tasks can be modified afterward. While there is no real task orchestration possible in PaROS, designing PaROS programs for specific tasks can be achieved in a compositional manner. The example applications the literature provides for PaROS include a Distributed Surveillance scenario only. There is no feature providing situation-awareness and run-time task generation. PaROS programs must be completed at design-time without a possibility for run-time adaptations. PaROS programs are programmed in code-style requiring programming knowledge which reduces their accessibility for non-technical experts.

4.5.5.9 Concluding the Comparison

Compared to the other approaches for task-orchestration and ensemble programming we analyzed in this section regarding their expressiveness, Maple is the only one providing a graphic

programming interface. While programming Ensemble Programs with MAPLE still requires a programmer to have some background in data handling and the definition of logical expressions it is the only approach not requiring in detail coding knowledge for generating Ensemble Programs. Thereby, MAPLE offers all task orchestration primitives that are necessary for designing complex programs. Compared to the aggregated possibilities other approaches offer, MAPLE only lacks control structures for defining time-dependent control flow. While there are some other approaches providing solutions for programming even greater sized systems than we do with MAPLE, approaches aiming at the same or similar applications than we do typically deal with systems of the same size. This comes from the fact that involving very high numbers (like, e.g., Meld [Ashley-Rollman et al., 2009] does) in applications aiming at real-world usage are not practicable and do require to much maintenance overhead. Some approaches for task orchestration also provide the possibility for run-time task generation. But there is no other recent approach using the possibilities that automated planning provides for dealing with this issue. Moreover, no other approach than MAPLE we analyzed in this section provides support for programming individual agents, teams of agents, and swarms of agents at the same time. Also, MAPLE is the only approach leaving as much flexibility in the concrete allocation of defined tasks to executing agents by using concepts like the Any-Agent, the All-Agent, the Set-Agent, and the Swarm-Agent. When focusing approaches for task orchestration applied to MAS/MRS, MAPLE is the only one providing support for encapsulating goal-oriented swarm behavior in a Collective Capability. Finally, MAPLE is the only approach for ensemble programming exploiting the flexibility in program design generated by decoupling agents from their capabilities. Recapitulating the findings in the literature, we can see that there is no approach for task orchestration supporting all features required for such, in our opinion. While all presented approaches deliver benefits for programming collectives, each lacks some aspects that are of great relevance from our point of view and that we support in MAPLE.

4.6 Future Research Directions

Possible future improvements for MAPLE potentially offering new research directions include an improved handling of data when programs command concurrent plan execution, i.e., triggered implicitly by replanning or explicitly by respective definitions for CNs in a HTN. Currently, concurrent execution of plans can lead to data inconsistency when not locking data stored in the shared world state accessed in multiple plans. We can find a promising option for improving on that state within the current progress achieved in distributed and synchronized databases, e.g., the approach of CockroachDB [CockroachLabs, 2021]. Integrating such technology into our approach has the potential for solving the named issues.

Further, the accessibility of MAPLE can be increased to allow also non-technicians the possibility for programming ensembles more easily. This would provide a real benefit, e.g., for letting rescue forces use MAPLE in the real-world to improve their daily business. Doing so can also include convenient user guidance while programming, e.g., to reduce programming errors like infinity loops, unreachable code, or inconsistent data usage and manipulation.

Moreover, time-dependent events can be an improvement for the current expressiveness we provide with maple. Letting an ensemble programmer include time-related conditions in ensemble programs, even more, complex programs can be generated for situations requiring such patterns.

Chapter Summary and Outlook

In this chapter, we described our approach for a Multi-Agent Script Programming Language for Ensembles (Maple). With Maple, we allow an ensemble programmer to define requirements for an executing system with the concepts of Hierarchical Task Networks (HTN) in a graphical way. In Maple, we adapt the concepts of HTN enabling an ensemble programmer to define complex ensemble programs involving sequential, parallel, conditional, concurrent, and repeated program control flow. We further provide abstractions for ensembles in Planning-Agent-Groups an ensemble programmer can use for the more accessible and more flexible instruction of ensembles. Combined with the possibility for instructing swarms of agents in addition to individual agents and teams thereof, we provide a maximum of flexibility to the system executing the program: A programmer does not need to decide on *how and in which configuration* to fulfill the defined requirements at design-time. Instead, these decisions are left to be made by the system at run-time. Because in Multipotent System, the association between capabilities and agents that can execute them is not of permanent nature but can be adapted in a self-organizing manner. The programmer does not need to take into account the respective configuration of agents. This allows an ensemble programmer to program with fewer restrictions on the one hand and, on the other hand, provides more flexibility to the Multipotent System for deciding who and in which configuration to execute the ensemble programs at run-time. Our evaluations show the expressiveness of Maple by example, showcasing all possible ways of instructing ensembles which we also support with complex examples for our case studies. Further, by comparing the expressiveness of Maple to that of other approaches for ensemble programming and task orchestration, we find that Maple is the only current approach providing all required features from our point of view.

In the following, we now describe how to form appropriately equipped ensembles in Multipotent System that can handle so created ensemble programs at run-time within Chapter 5 and Chapter 6. In the subsequently following Chapter 7 we then further describe how to execute so defined ensemble programs.

Chapter 5

A Self-Organization Mechanism for Ensemble Formation

Summary. As the result of ensemble programming we described in the last chapter, we derive a plan from the ensemble programmer's definitions within an Hierarchical Task Networks (HTN). Besides information concerning the execution of this plan encoded in the ensemble levl part and the agent level parts of a Ensemble Program (which we focus on in Chapter 7), this plan also contains the requirements to an ensemble that have to be satisfied before executing the Ensemble Program. Requirements define the number of agents for working on a plan, and the respective capabilities each of these agents must provide for participating in the ensemble. In this chapter, we investigate the problem of forming appropriate ensembles for such plans within a Multipotent System so that all requirements defined by teh plan are satisfied. We find that this problem is an instance of the Task Allocation Problem, which we can solve by coalition formation, as the literature proposes. We adopt a distributed and market-based approach that is familiar from Multi-Agent Systems (MAS)/Multi-Robot Systems (MRS) to solve coalition formation in Multipotent Systems systems. Doing so allows us to cope with the most important properties of Multipotent Systems that include a distributed deployment, possible high numbers of agents involved, and their inherent property of frequently changing the compositions of their agents' capabilities. That way, we can integrate the agents' ability for self-aware reasoning on their respective task qualification. For determining valid solutions to the Task Allocation Problem, we formulate it as a Constraint Satisfaction (and Optimization) Problem (CSOP) which we can model with the constraint modeling language MiniZinc. To demonstrate the feasibility of using our approach with flying ensembles, we evaluate the scalability of deploying it to real hardware we can use within these flying ensembles. Further, we investigate the possibilities of integrating optimization criteria in the satisfaction problem of task allocation enabling the Multipotent System to find the best composition for each ensemble. Moreover, we give proof of concepts for deploying our approach of market-based task allocation to real-world flying ensembles in a set of multiple indoor laboratory experiments.

Publication. Contents of this chapter have been published in [Kosak et al., 2016a,b, 2018, 2020b].

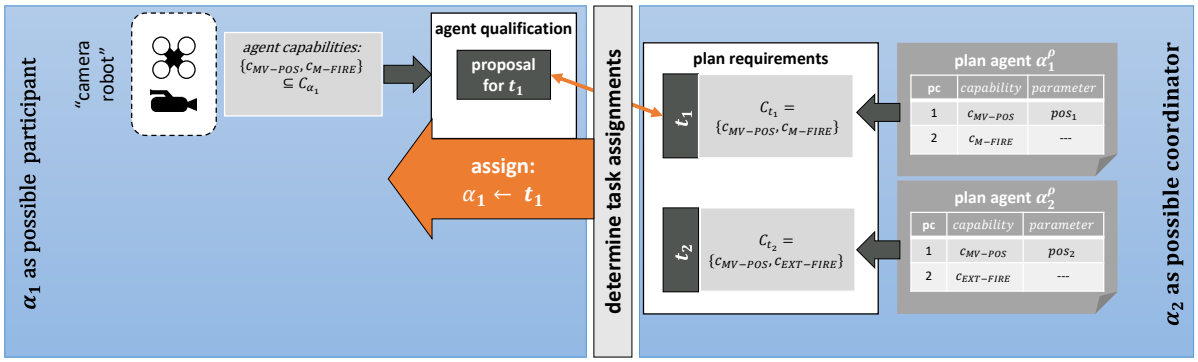
5.1 Self-Awareness Based Ensemble Formation in Multipotent Systems

When designing SCORe missions, the user of a Multipotent System does this by defining which capabilities agents should execute in which logical order and in which situation for achieving the goals of the specific SCORe mission at a later point in time. Often these definitions do not involve only one but multiple agents the user needs to cooperate in the SCORe mission. By doing so, the user implicitly defines requirements for an ensemble of agents that needs to be formed within the Multipotent System at run-time: By performing automated planning on the user's definition of the SCORe mission, a plan ρ is generated for the current conditions the Multipotent System is currently situated in. This plan then specifies how many agents are required to cooperate and which capabilities those agents individually need to provide for this cooperation. In addition, each plan ρ also contains information on the pattern of this cooperation, i.e., how and when individual agents should execute the enlisted capabilities and how to exchange information with each other. While we concentrate on the details of the different facets of this cooperation in Chapter 7, we focus on forming appropriate ensembles $\mathcal{E}^\rho$ for a plan ρ in this chapter first. The challenge in forming such ensembles is twofold:

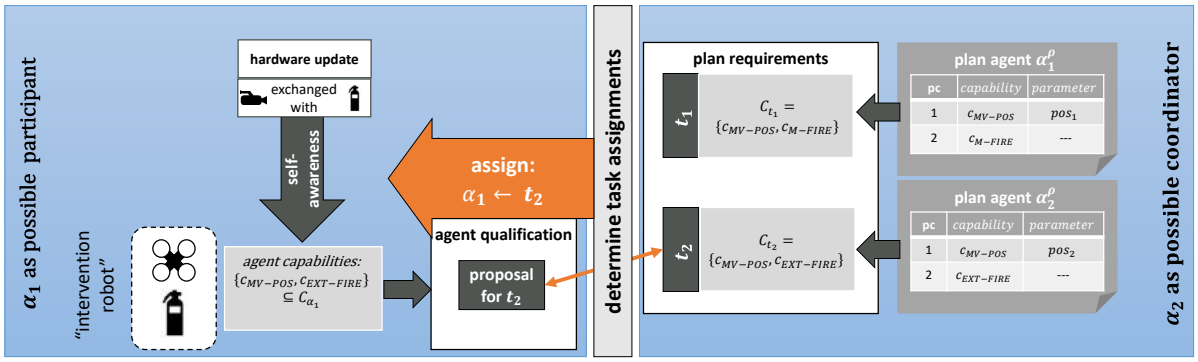
First, we need to specify the requirements for agents in a form that can be further processed automatically by the system. To specify the number of agents we require in $\mathcal{E}^\rho$ and express the requirements for each of these agents individually, it is common to specify a task t for every Planning-Agent named in the plan [Gerkey and Matarić, 2004]. This results in a set of tasks $\mathcal{T}^\rho$ for the plan ρ. Each of these tasks $t \in \mathcal{T}^\rho$ further defines the requirements an agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ performing t needs to fulfill. In our scenario, these requirements are described by a set of required capabilities $\mathcal{C}_t$ an agent must provide for a valid assignment.

Second, we need to find appropriately equipped agents that cooperatively can fulfill all of the requirements defined by ρ. Therefore, we need to be aware of the set of capabilities $\mathcal{C}_\alpha$ each agent can provide. In a Multipotent System $\mathcal{MS}$, the information describing these sets for all agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ is not static but can dynamically be modified by adaptations in the agents' hardware configuration $\mathcal{SDH}_\alpha$. According to its respective configuration, the agent can qualify for different or multiple tasks $t \in \mathcal{T}^\rho$ when considered for participating in a plan ρ at different times. When forming an ensemble $\mathcal{E}^\rho$ for ρ, we must first update our knowledge on the different agents' current set of provided capabilities $\mathcal{C}_\alpha$ before deciding which agent should perform which task. Moreover, we need to ensure that we assign each task only to the required number of agents specified by ρ.

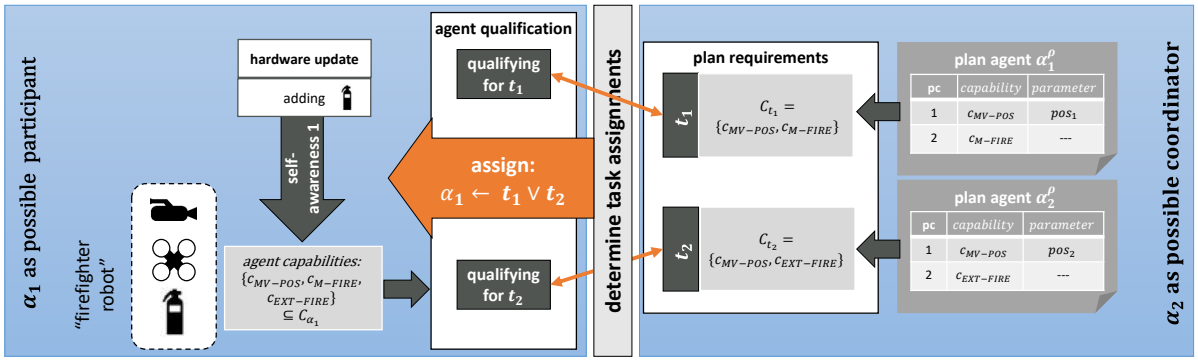
We illustrate the problem with an example from our case study on *Dealing with Forest Fires* and describe the different situations for an individual agent α_1 in Figure 5.1. In a situation where we already have detected a fire at position pos_2 we need to extinguish and have a suspicion of another fire at position pos_1, a user-defined plan ρ_{FIRE} might require two agents to fill the place-holding roles of the Planning-Agents α_1^ρ and α_2^ρ to cooperate. In ρ_{FIRE}, the first Planning-Agent α_2^ρ should extinguish the known fire at pos_1 with a capability $c^p_{\text{EXT-FIRE}}$ and the second Planning-Agent α_1^ρ should measure whether there is another fire present at pos_2 by executing the capability $c^p_{\text{M-FIRE}}$ there. We encode these requirements from ρ^{FIRE} in two tasks $\{t_1, t_2\} = \mathcal{T}^{\rho_{\text{FIRE}}}$ and thus need to find two different agents $\alpha_{i,j} \in \mathcal{A}_{\mathcal{MS}}$ providing the required capabilities allowing us to assign t_1 and t_2 to them. In a configuration like we depict it in Figure 5.1a, an agent α_1 provides all requirements for t_1. As it is configured as a "camera robot" carrying a camera module with an UAV, α_1's set of capabilities $\mathcal{C}_\alpha$ contains



(a) possible task assignment for agent α_1 before updates happen to its hardware configuration $\mathcal{SDH}_\alpha$



(b) possible task assignment for agent α_1 after exchanging hardware in the configuration $\mathcal{SDH}_\alpha$ influencing $\mathcal{C}_\alpha$



(c) possible task assignments for agent α_1 after adding hardware to the configuration $\mathcal{SDH}_\alpha$ influencing $\mathcal{C}_\alpha$

Figure 5.1: Possible assignments of tasks during Ensemble Formation for a plan ρ_{FIRE}, focusing on α_1 as a possible participating in ρ and α_2 as a possible coordinator of the process. During that process agent α_2 aims at forming an ensemble $\mathcal{E}^\rho$ properly compiled for working on ρ_{FIRE} whose cooperative execution it then can coordinate as *coordinator*. Agent α_1 aims at *participating* in $\mathcal{E}^\rho$ as an executing instance. Figures 5.1a to 5.1c show three different possible results from that process, depending on α_1's hardware configuration that influences its provided capabilities.

all required capabilities of t_1, i.e., $\mathcal{C}_{t_1} = \{c^p_{\text{MV-POS}}, c^p_{\text{M-FIRE}}\} \subseteq \mathcal{C}_{\alpha_1}$. Thus, we can assign t_1 to α_1 in that configuration. If we change the hardware composition of α_1 by exchanging the camera module with an fire extinguisher module (cf. "intervention robot" in Figure 5.1b), the set of provided capabilities for α_1 changes to $\mathcal{C}_{\alpha_1} = \{c^p_{\text{EXT-FIRE}}, c^p_{\text{MV-POS}}\}$, i.e., α_1 loses its previously provided capability $c^p_{\text{M-FIRE}}$. Consequently, α_1 no longer qualifies for t_1 but for t_2 instead, i.e., $\mathcal{C}_{t_2} = \{c^p_{\text{MV-POS}}, c^p_{\text{EXT-FIRE}}\} \subseteq \mathcal{C}_{\alpha_1}$. Thus, after performing such adaptation to α_1's set of provided capabilities, we now can assign t_2 to α_1. If, instead, we let the camera module last within α_1's configuration when adding the fire extinguisher module (cf. "firefighter robot" in Figure 5.1c), we increase the set of α_1's provided capabilities $\mathcal{C}_{\alpha_1}$ to $\{c^p_{\text{MV-POS}}, c^p_{\text{EXT-FIRE}}\}$. Then α_1 qualifies for both, t_1 and t_2 so that we can assign both of these tasks to α_1. Because we assume that each agent can only assign one task at a time (i.e., while we can assign $\alpha_1 \leftarrow t_1 \vee t_2$, we need to decide for one of them when allocating tasks to agents), we thus require another agent $\alpha_{i\neq 1}$ for assigning the respective other task. Thus, we need to evaluate the configurations of all other agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$, determine their current qualifications for the relevant tasks, and decide on which agent finally performs t_1 and t_2 from the respective sets of agents providing all required capabilities.

The underlying problem we need to solve when forming ensembles is an instance of the Task Allocation Problem. In the following, we now first investigate its definition and classify the problem of Ensemble Formation in Multipotent Systems as such Task Allocation Problem following the common taxonomy defined in the literature in Section 5.2. We then investigate related work concerning the problem of task allocation in Section 5.3. Further in Section 5.4, we propose our approach for solving the Task Allocation Problem with a self-aware, distributed, and market-based mechanism. Finally, we investigate the feasibility and scalability of deploying our approach to real hardware we can use with flying ensembles that are subject to potentially lower computational power when compared to other computation hardware in Section 5.5.

5.2 Ensemble Formation as an Instance of the Task Allocation Problem

While planning and scheduling as we perform them in Chapter 4 focuses on the questions *what actions* we need to be executed by a system and *when* we need them to be executed, task allocation concentrates on the question *by whom* actions should be performed [Russel and Norvig, 2014]. This decision then is often to be made consistent also with resource constraints [Bartusch et al., 1988]. When mapping Task Allocation Problems (and task scheduling problems) to MAS/MRS, a categorization of the problem as an instance of the MRTA according to the taxonomy of Gerkey and Matarić [2004] is common. Therefore, we need to classify the way we allocate tasks in our application regarding three parameters of relevance, i.e., we need to determine whether

1. each agent participating in the process of allocation can deal with multiple tasks (MT) or just a single task (ST) at the same time.
2. tasks need exactly one single robot (SR) or multiple robots (MR) to be accomplished.
3. the allocation of tasks is either to be made instantaneously (IA) or time-extended (TA), i.e., classify the problem as a task assignment (IA) or task scheduling (TA) problem.

To classify the problem of Ensemble Formation, we need to handle within Multipotent Systems according to this taxonomy, we briefly recapitulate the properties of the problem.

1. In SCORe missions and plans derived from them with automated planning (cf. Chapter 4), we expect locations where agents need to execute their tasks to be spatially distributed (cf. Chapter 2). Thus, we consider agents that work within only one of these plans ρ simultaneously, i.e., deal with single-task robots (ST).
2. Because we try to find an ensemble $\mathcal{E}^\rho$ for each plan ρ by solving an instance of the Task Allocation Problem, we deal with multi-robot tasks (MR) in our approach. This is because we consider *a single plan ρ to be a single task* in that terminology.
3. Because we handle the scheduling of tasks during automated planning on a user-defined problem domain description, we do not need to take into account timely and logical order during task allocation. Instead, we require to assign the tasks during task allocation as soon as they appear in the system, i.e., we require instant assignments (IA).

Subsuming this analysis, we can define the problem of Ensemble Formation to be an instance of the MRTA we can classify as ST-MR-IA. This classification also helps us to find candidate strategies for solving it in the broadly related literature on task allocation.

In general, the challenge in such strategies for forming ensembles $\mathcal{E}^\rho$ is twofold. On the one hand, we need to answer the *first question*: "Which agent qualifies for performing which of the different tasks $t \in \mathcal{T}^\rho$ from the plan ρ?". Because each task addresses one or multiple capabilities that need to be executed for reaching the goal of the plan, we can define the requirements for performing a task t in a set of capabilities $\mathcal{C}_t$. In the context of Multipotent System, this can include physical capabilities directly originating from different SDH and virtual capabilities that can be dependent on combinations of different SDH. On the other hand, for each task $t \in \mathcal{T}^\rho$, we then need to answer the *second question*: "Which concrete agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ does finally perform a tasks $t \in \mathcal{T}^\rho$ from the plan ρ out of the set of those agents fulfilling all requirements of t?". Throughout answering questions one and two, we need to take into account the particular requirements of plans defined with our approach for designing SCORe missions in Chapter 4 using MAPLE. There, plans can address Identified-Planning-Agents $\alpha^\rho{}_i \in \mathcal{A}_I^\rho$ and unidentified agents α^ρ like the All-Agent $\alpha_\forall^\rho$, the Any-Agent $\alpha_\exists^\rho$, and the Swarm-Agent $\alpha_{\{\substack{\text{MIN}\\\text{MAX}}\}}^\rho$. This fact further complicates the problem of Ensemble Formation, which we need to respect when formulating it as an instance of the Task Allocation Problem.

5.3 Related Work

Because the allocation of tasks (i.e., $t \in \mathcal{T}^\rho$ from a plan ρ) depending on resources of potential task assignees (i.e., agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ providing certain capabilities $c \in \mathcal{C}_\alpha$) is of central importance in our approach, we investigate other approaches tackling the Problem of Task Allocation for MAS/MRS in the following. In general, tasks may need to be performed at a specific time while the total make-span, i.e., the time needed to reach the goal state, must be minimized. This problem stems from the field of operations research [Pinedo, 2016] and is called the Job-Shop Scheduling Problem (JSP) [Lawler et al., 1993]. When considering realistic settings, where scheduled tasks also entail specific resource demands (like processing power, knowledge, capability, information, or expertise [Weiss, 2001]), the complexity of scheduling becomes NP-hard [Russel and Norvig, 2014]. To deal with this difficulty, heuristics are often used (e.g., minimum slack [Smith and Cheng, 1993], ant system [Colorni et al., 1994]). Other methods increase the flexibility of resource schedules [Muscettola, 2002] or map scheduling problems to Constraint Satisfaction (and Optimization) Problem (CSOP) to exploit (resource) constraint propagation [Laborie, 2003] for increasing the performance in finding solutions to the problem.

In Section 5.2, we analyzed that when searching for solutions to the Problem of Ensemble Formation, we can focus on the Task Allocation Problem. We can do so as, at this stage, the user already scheduled individual actions and defined requirements concerning resources (cf. Chapter 4). We can classify solutions to the Task Allocation Problem described in the literature into two general strategies, depending on the system class they are applied to: Are agents configured heterogeneously or homogeneously, i.e., can all of them provide the same capabilities, or do they differ [Zlot and Stentz, 2006]? With Multipotent System, we introduced a new system class aiming to combine the benefits of homogeneous and heterogeneous systems. Thus, we need to carefully determine which solutions from literature we can map to our problem of Ensemble Formation. Concerning the Problem of Task Allocation, we decide that those approaches focusing on heterogeneous systems are more relevant for application in Multipotent System. We come to this decision because we cannot assume to have a homogeneously configured system every time the Multipotent System needs to form a new ensemble but instead generate homogeneity first when we require it according to the user-defined requirements. After all, a new plan including specific tasks can become relevant and thus our solution for Ensemble Formation must also deal with such plans. Nevertheless, in the following analysis of existing literature on handling the problem of Task Allocation, we briefly also investigate solutions available for homogeneous systems.

In purely homogeneous systems, approaches for solving the Task Allocation Problem often adopt behavior from nature, e.g., in the form of swarm algorithms. These solutions tend to be suboptimal [Zlot and Stentz, 2006], which still can be sufficient for the considered system. In [Khaluf, 2016], e.g., the Task Allocation Problem is solved by applying a probabilistic approach adopted from natural swarms for constructing tasks with a swarm of simple robots. While the system can solve the task allocation problem in this setting, it is thus limited to allocating only the one type of task that is relevant for their use case, making it hard to be applied when task requirements frequently change like we assume them to do in SCORe missions. Jevtic et al. [2012] use a behavior adopted from bees for achieving task allocation in a swarm of robots. Also, in this research, the tasks required to be solved by the system have equal requirements reducing its adaptability for scenarios where these requirements differ from task to task. In [Brutschy et al., 2014], the authors extend possible requirements to two different tasks settled in a harvesting task. In their approach, members of a robot swarm decide locally which tasks they should assign depending on the specific conditions. They build their approach on the two assumptions that robots can handle each of the tasks that can become relevant and that these tasks do not vanish after one robot successfully finished working on it but instead, there is an infinite source of tasks. The assumptions they make do not cope with those we assume for our setting. Because most other approaches dedicated to homogeneous systems rely on such or similar assumptions made to tasks or robots, we find that we cannot apply them for the type of Task Allocation Problem we need to handle in Multipotent System.

Instead, we focus on approaches for solving the Task Allocation Problem in heterogeneous systems. Those are either of centralized or of distributed nature. Centralized approaches focus on finding optimal solutions to the Task Allocation Problem [Khamis et al., 2015; Mosteo and Montano, 2010]. Because such approaches rely on always up-to-date data concerning the involved agents' internal states, pure centralized approaches can only be applied when ensuring this condition is feasible every time an allocation should be found. An example for such system is that Bowling et al. [2004] propose a central approach for handling the Task Allocation Problem for. They apply it for a robot soccer team where they can aggregate all relevant information with a surveying camera system. Also, purely simulated environments employ

centralized approaches since all relevant information is inherently available [Rosencrantz et al., 2003; Amigoni et al., 2005]. Sometimes, also real-world systems make use of centralized approaches for solving the Task Allocation Problem. Liu and Kroll [2012], e.g., apply it to the use case of industrial plant inspection. Also, Higuera and Dudek [2013] use a central approach for task allocation applying findings of game theory to a team of robots for achieving a fair distribution of tasks in their system. While the presented collection of approaches is only an excerpt of those approaches for solving the Task Allocation Problem centrally, applying such for agents acting in the real world tends to be less followed recently – as a direct consequence of the inherent difficulties centralized approaches have for solving NP-hard problems. This comes from the drawbacks such centralized approaches underlie in general. Using a central instance (1) often becomes a bottleneck when scaling the problem size, (2) can be safety-critical as they quickly produce a single point of failure, and (3) can require frequent communication in distributed systems that can become inefficient [Khamis et al., 2011]. These properties make central approaches less interesting for applying them to the problem of Ensemble Formation we face in Multipotent System.

Instead, decentralized or distributed approaches for solving the Task Allocation Problem from the literature are more appealing. Some recent approaches focus on distributing the required computational resources for calculating a solution to the Task Allocation Problem, e.g., with decentralized genetic algorithms [Iordache et al., 2007; Choi et al., 2011; Patel et al., 2020] or by using distributed constraint programming [Choi et al., 2010; Fioretto et al., 2018]. While the idea of problem decomposition can reduce the locally required computation times per node, for merging and combining efforts, a high number of messages need to be exchanged. Other approaches, e.g., [Dorigo et al., 2013], use scripted behavior for solving the Task Allocation Problem, making them uninteresting for dynamic task environments. The most common paradigm for solving the Task Allocation Problem in a decentralized fashion is that of market-based mechanisms [Dias and Stentz, 1999; Dias et al., 2006; Anders et al., 2015] that rely on strict communication protocols (such as the contract net protocol [Smith, 1980]) when direct interaction between robots/agents is possible. While market-based approaches suffer some from possible sub-optimality of solutions and the mechanisms can be exploited in competitive scenarios by strategical voting [Weiss, 2013], the benefit of such market-based approaches is clear compared to the drawbacks we can find for centralized approaches, especially when assuming the participants act benevolently. Instead of creating a central bottleneck (1), market-based approaches can exploit the possibility of decomposing the Task Allocation Problem and distributing its calculation to participants [Khamis et al., 2015; Dias et al., 2006; Zlot and Stentz, 2006]. Compared to the single-point-of-failure in central approaches (2), market-based approaches inherently can absorb failures of individual participants [Khamis et al., 2015; Dias et al., 2006; Zlot and Stentz, 2006], e.g., using appropriate strategies with non-fixed auctioneers like proposed by Coltin and Veloso [2010]. Further, instead of producing significant communication overhead by keeping information concerning all agents in a system up-to-date at a central instance (3), market-based approaches only aggregate information on the participants changed conditions when this is relevant for solving a specific instance of the problem [Khamis et al., 2015; Dias et al., 2006; Zlot and Stentz, 2006].

All current approaches have in common that they do not consider the possibility of reallocating capabilities to agents, which we emphasize in Multipotent System. In our approach, we want to improve the current state of the art by harnessing this neglected potential. Thus, we need to find such a solution for the Task Allocation Problem that can also cope with frequently changing conditions concerning tasks, their requirements, and the configuration of agents we

need those tasks to be assigned to. We, therefore, propose to adapt the idea of market-based task allocation known to perform sufficiently in distributed systems and combine this with a self-awareness mechanism enabling agents to participate in the process with always up-to-date information.

5.4 An Approach For Self-Aware, Distributed, and Market-Based Ensemble Formation (SELF-MADE)

The main properties of Multipotent Systems that affect the decision for a mechanism dedicated to tasks allocation are the distribution of Multipotent Systems to multiple nodes in a network, the potential scale concerning agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ that a Multipotent System might involve, and the flexibility in the set of capabilities $\mathcal{C}_\alpha$ each of these agents can provide at different times in a Multipotent System uptime.

Considering these properties, an answer to the first question we define in Section 5.2 can be generated best by the individual agents themselves. Agents are experts concerning their current provided capabilities $\mathcal{C}_\alpha$, i.e., always have up-to-date information on their local hardware configuration $\mathcal{SDH}_\alpha$ affecting the composition of $\mathcal{C}_\alpha$. For answering the second question formulated in Section 5.2, we need an instance in the Multipotent System able to aggregate the answers given to question one by the individual agents. Only such instances can assure that each task $t \in \mathcal{T}^\rho$ is assigned to an agent. As discussed before, the best approach for achieving this is that of a market-based approach for task allocation [Dias et al., 2006]. In general, market-based task allocation is considered to provide beneficial properties we want to profit from in our approach for Ensemble Formation, e.g., the efficiency of the task allocation, robustness, and scalability Khamis et al. [2015]. In a market-based approach, we let agents respond to a Call for Proposals (CfP) that an auctioneer initiates. While a CfP can contain the requirements of the tasks $t \in \mathcal{T}^\rho$, proposals in return indicate whether the respective agent can meet them. The main benefits of a distributed market-based approach compared to a centralized approach when applied for task allocation in Multipotent Systems thus are:

- We can avoid the frequent communication of updates in agent configurations and restrict them to only those situations where communication is necessary. Such a situation occurs in Multipotent Systems when a Task Allocation Problem needs to be solved, i.e., when a new plan is derived by automated planning on PLAN LAYER (cf. Section 3.2.5.
- We can handle the possible scale of agents involved in our system because agents for themselves evaluate their qualification, which can be nontrivial (cf. our proposed handling of virtual capabilities on SEMANTIC HARDWARE LAYER we describe in Section 3.2.8).
- We can handle possible agent failures restricting their qualification for a task. That way, we can absorb failures affecting whole agents or only parts of their hardware configuration. If an agent is no longer available, it simply does not respond to a CfP and thus does not send a proposal to the auctioneer. Suppose any hardware component failure within the agent's configuration affects the capabilities required in tasks included in the CfP. In that case, the agent also can respect this when generating a proposal, i.e., only propose to be qualified for those tasks $t \in \mathcal{T}^\rho$ it still provides all required capabilities for despite the hardware failure.

In a centralized approach, we would lose many of these beneficial properties:

1. We would need to synchronize information every time a configuration of an agent changes. This can become inefficient when the number of agents increases or changes in the hardware configuration of agents happen frequently.
2. Handling large sets of agents can become inefficient when dependencies between hardware configurations and capabilities are nontrivial. If these differ from agent to agent, a central instance would need to know and evaluate all these individual dependencies, increasing time spent on computation to an unacceptable level.
3. If failures occur for individual agents, a central instance needs to be updated. While this can still be handled if only individual hardware from an agent's configuration is affected, e.g., with an update similar to that when changing the configuration intentionally, handling complete failures of agents causes problems. Because a central instance would not know that an agent is unavailable any longer after such a failure, it would require deriving information on the agents' current states before each task allocation. Then, such a central approach would soon become very similar to a market-based approach that explicitly performs such updates in a CfP.

These considerations confirm our opinion that a market-based approach is appropriate for solving the Task Allocation Problem, which we instantiate for forming ensembles for plans in Multipotent System. We thus propose an approach for a Distributed, Self-Aware Market-Based Mechanism for Ensemble Formation (SELF-MADE).

5.4.1 Technological Background

We formalize the task allocation problem as an instance of the Constraint Satisfaction (and Optimization) Problem (CSOP). Doing so enables us to perform the selection of proposals to assure that the resulting allocation respects all defined task requirements. This means we allocate each task to an agent that is also capable of executing it concerning

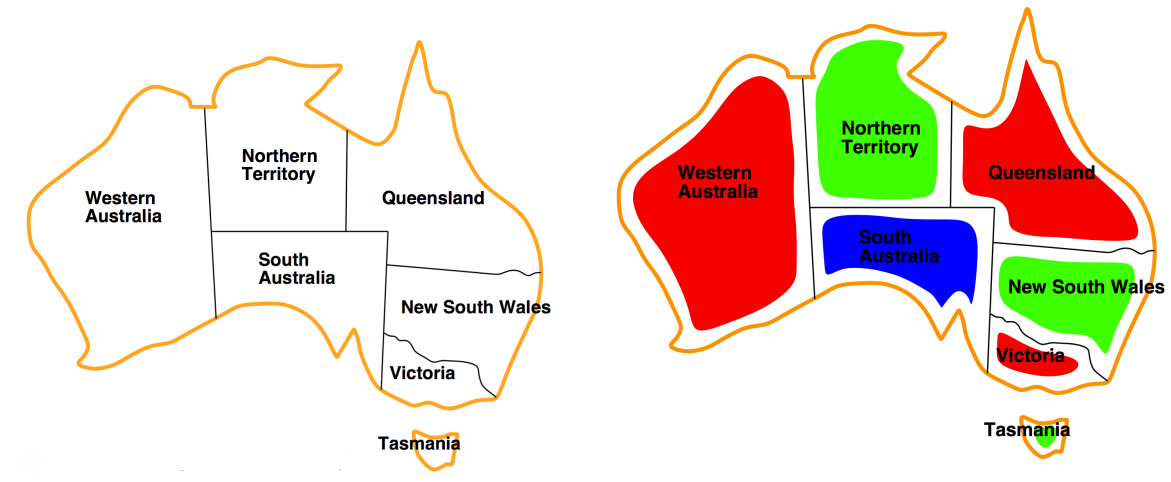
- the task's required capabilities,
- the agent's provided capabilities, and that
- each agent does only execute one task at a time (cf. our definition of the underlying MRTA as ST-MR-IA in Section 5.2).

For modeling the selection process as a CSOP, we use the MiniZinc constraint modeling language [Nethercote et al., 2007]. We execute our evaluations concerning scalability with a CSOP we model in MiniZinc. Thus, we now first briefly introduce the theoretical background of CSOP and how we can model and solve CSOP with MiniZinc before we second present our solution to the specific Task Allocation Problem of Multipotent System we introduced in Section 5.2.

5.4.1.1 Constraint Satisfaction and Optimization Problems

The Constraint Satisfaction (and Optimization) Problem (CSOP) defines a class of combinatorial problems that are already successfully solved today in numerous domains, such as operations research or resource allocation problems [Apt, 2003; Tsang, 1993]. Tsang [1993] defines the problem class of CSOP as follows:

> "Basically, a CSP is a problem composed of a finite set of variables, each of which is associated with a finite domain, and a set of constraints that restricts the values the variables can simultaneously take. The task is to assign a value to each variable satisfying all the constraints."



(a) initial conditions (b) possible solution

Figure 5.2: The Map Coloring Problem instantiated for the the different states of Australia.

Mathematically, the CSOP thus is defined as a triple *(Z, D, C)* as follows [Maher and Puget, 2003; Tsang, 1993]:

- Z is a finite set of variables $Z = \{x_1, \ldots, x_n\}$
- D (domain) is a function that maps each variable in Z to a set of objects of arbitrary type $D = \{D_1, \ldots, D_n\}$ with $\{x_1 : D_1, \ldots, x_n : D_n\}$
- C is a finite set of constraints $C_j(Z_j)$, $j \in \{1, \ldots, m\}$, which is any subset of variables $Z_j = \{z_{j_1}, \ldots, z_{j_k}\} \subseteq Z$ set in relation to each other. Valid combinations are thereby given as a subset of $D_{j_1} \times \cdots \times D_{j_k}$.

A solution of an CSOP then is defined by a valid assignment for all variables in Z with values from their domains D that satisfy all constraints in C [Tsang, 2014]. Each instance of the CSOP can be classified into one of the following three categories depending on the requirements. Problems of the first category search for a single solution to the CSOP. Problems of the second category search for *all* solutions to the CSOP. Problems of the third category then search for an *optimal* solution to the CSOP from the set of solutions, i.e., introduce the aspect of *optimization* to the problem according to an objective function mapping each solution to an ordered set, typically the real numbers. A well-known example we use illustrate the use of CSOP is the so-called "map coloring problem"which asks to color each region on a map either red, green, or blue under the constraint that no neighboring regions receive the same color [Russel and Norvig, 2014]. A commonly used example application of this problem in the literature is the problem of coloring a map of the country Australia and its states (cf. Figure 5.2) [Russell and Norvig, 2015]. The CSOP for this problem can be modeled as follows:

- $Z = \{WA, NT, SA, Q, NSW, VIC, T\}$
- $D_z = \{red, green, blue\}$
- $C = \{WA \neq NT, NT \neq SA, \ldots\}$

The set of variables (Z) contains all states of the country, e.g. *WA* stands for Western Australia. The domain (D) restricts possible assignments for all variables in Z to a value from the set $\{red, green, blue\}$ The constraints (C) define that no neighboring regions get the same color,

```
1  include "alldifferent.mzn";
2  % problem data
3  int: n;
4  set of int: ROBOTS = 1..n;
5  int: m; set of int: TASKS = 1..m;
6  array[ROBOTS,TASKS] of int: profit;
7  % decisions
8  array[ROBOTS] of var TASKS: allocation;
9  % have robots work on different tasks
10 constraint alldifferent(allocation);
11 % goal
12 solve maximize sum(r in ROBOTS)(profit[r, allocation[r]]);
13 output["allocations = ", show(allocation), "\n",]
14
```

Listing 5.1: MiniZinc Code

Figure 5.3: Minimal example of a MiniZinc constraint model for allocating tasks to robots.

e.g. Western Australia does not get the same color as the Northern Territory. A valid solution for the problem is, e.g., $WA = red, NT = green, Q = red, NSW = green, VIC = red, SA = blue, T = green$ (cf. Figure 5.2b). For calculating the solution to this and other problems, we can model the respective instance of the CSOP, e.g., with the constraint modeling language MiniZinc that provides access to efficient solvers for that purpose.

5.4.1.2 The Constraint Modeling Language MiniZinc

For solving a CSOP, there exists a variety of solution strategies and solvers such as CPLEX or Gecode [Nethercote et al., 2007]. However, these solvers mostly use incompatible and different modeling languages, making it difficult for users to exchange solvers if required, e.g., because of their differing efficiency in different problem classes. For this reason, Nethercote et al. [2007] developed the expressive yet straightforward modeling language MiniZinc dedicated to constraint programming. MiniZinc provides a reasonable middle ground for exploiting the different advantages of the numerous solvers. The developed modeling language MiniZinc can be used to model optimization and satisfaction problems over both integers and real numbers. The modeling of a MiniZinc problem consists of two parts. The model, which describes the structure of a problem, and the data, which defines the concrete problem instance in more detail. The model and data can be defined in separate files if required for modularity. Each MiniZinc model is defined by a sequence of elements that can be in any order [Nethercote et al., 2007]. In the following, we give a brief insight into the syntax of MiniZinc using an example. The example is a simplified version of a Task Allocation Problem, intending to assign one out of m tasks to each of n available robots while maximizing the total profit summed up over all robots.

In Line 1 of Listing 5.1, we use the *include* command to reference the contents of another model, such as the model *"alldifferent.mzn"*. That way, we can use efficient implementation of algorithms from external libraries, particularly dedicated domain filtering algorithms known as constraint propagators. The general form for doing so in MiniZinc is using statements of the form *include* ⟨*filename*⟩*;*. In Line 3 of Listing 5.1, we use the expression *int:* to declare a

Table 5.1: Matrix indicating the profit an allocation of a certain task to a robot has.

	t1	t2	t3	t4
r1	4	3	1	7
r2	6	4	1	2
r3	7	1	1	3

variable n of type integer we use to indicate the number of robots involved in the following. The basic parameter types available in MiniZinc are integers (*int*), floats (*float*), booleans (*bool*), and strings (*string*). The general form is of declaring variables in MiniZinc is using statements of the form ⟨*variable*⟩*:* ⟨*name*⟩*;*. In Line 4 of Listing 5.1, we declare the new parameter *ROBOTS* within the MiniZinc model and define it to be a *set* with the length of n, defined by a *range expression {1,...,n}*. The general form for declaring sets like this in MiniZinc is using statements of the form *set of* ⟨*type-inst*⟩ ⟨*expr1*⟩ *...* ⟨*expr2*⟩*;*. In Line 6 of Listing 5.1, we declare an array of integer parameters called *profit* having two dimensions n and m with n set to the length of *ROBOTS* and m to the length of *TASKS*. The general form for such declarations in MiniZinc is using statements of the form *array[*⟨*index-set1*⟩*,..,*⟨*index-setn*⟩*] of* ⟨*type-inst*⟩*;*. In Line 8 of Listing 5.1, we define a one-dimensional array of *decision variables* called *allocation*. The array has the length of *ROBOTS* with a value from the range of 0 up the length of *TASKS*. In addition to that, we can express an allocation of a task to a robot by entering the index of the task at the respective robot's position in the array. In Line 10 of Listing 5.1, we define the only constraint necessary in this MiniZinc model illustrating one of MiniZinc's biggest benefits. It uses the *alldifferent* constraint whose implementation we included from another model in Line 1 of Listing 5.1. That way, we can compose complex models with high expressiveness and efficiency by exploiting the offered possibility of modularized programming and including functions from well-tested external libraries. *Alldifferent* provides a specialized implementation of a function that introduces constraints in the background, assuring that all entries in the given decision variable *allocations* must be different for a valid solution. For the MiniZinc model in Listing 5.1, we can assure that in a valid solution, each task gets assigned to only one robot. In Line 13 of Listing 5.1, we define the optimization goal of this MiniZinc model. With the keyword *solve*, we specify the class of problem we want to solve, i.e., either a satisfaction problem with *solve satisfy* or an optimization problem with *solve maximize* ⟨*arithmetic expression*⟩*;* or *solve minimize* ⟨*arithmetic expression*⟩*;*. In the MiniZinc model in Listing 5.1, we define an optimization problem that searches for a solution providing the maximum summed up profit possible with a task allocation of tasks to robots. In Line 13 of Listing 5.1, we finally can define the *output* of the model usable for further processing a valid solution. In this case, the array *allocation* is the part of the solution we are interested in. The general form to define the output of a MiniZinc model is a statement of the form *output[*⟨*string expression*⟩*, ...,* ⟨*string expression*⟩*];*.

As an example, we can instantiate the model with the variables n set to 3 and m set to 4, combined with a matrix expressing the profit each of the robots has when assigning a certain task to it (cf. Table 5.1). In Table 5.1, the profits for the individual tasks $(t_1, \ldots, t_4)$ and for the robots $(r_1, \ldots, r_3)$ are specified. Consequently, e.g., robot r_1 can achieve a profit of 4 with task t_1. The solution to the optimization problem defined in Listing 5.1 with the input from Table 5.1 then is a task allocation assigning $r_1 \leftarrow t_1$, $r_2 \leftarrow t_2$, and $r_3 \leftarrow t_4$.

MiniZinc is designed to solve the above example with different solvers efficiently. To do

this, it transforms the model instance (model and data) into a so-called FlatZinc model. This translation can be read by a solver that provides a FlatZinc interface [Nethercote et al., 2007]. Subsequently, the model can be solved by this solver using various search strategies, such as backtracking or heuristics [Meseguer et al., 2006]. This ensures a high degree of interchangeability and allows the advantages of each solver to be exploited, e.g., different search heuristics [Nethercote et al., 2007]. MiniZinc, with its simplicity, expressiveness, and ease of implementation, is thus a practical choice for modeling and solving CSOP as we consider them in our approach for task allocation in this chapter as well as our approach for resource allocation in Chapter 6.

5.4.2 Formalization of the Underlying Task-Allocation Problem

For realizing SELF-MADE, we formalize the problem in three stages in the following. First, we formalize the requirements of a plan as a task we include in a CfP for that plan. Second, we describe how agents receiving this CfP can analyze their qualifications concerning these requirements and formalize how they participate in the task allocation by generating proposals for the tasks in the CfP. Third, we formalize the Task Allocation Problem as an instance of the CSOP.

5.4.2.1 Defining Plan Requirements in a Call for Proposals (CfP)

Requirements for a plan ρ originate from the user's definition in the SCORe mission. When formalizing these requirements for a CfP, we thus need to take into account all capabilities $c \in \mathcal{C}$ associating any Planning-Agent $\alpha^\rho \in \mathcal{A}^\rho$ occurring in the plan ρ. Information from ρ that is of interest for generating a plan-specific CfP thus contains the following relevant capability-agent-associations: When programmed with MAPLE, capabilities occurring in ρ can either be physical capabilities $c^p_\Upsilon \in \mathcal{C}^p \subseteq \mathcal{C}$ or virtual capabilities $c^v_\Upsilon \in \mathcal{C}^v \subset \mathcal{C}$, each of different types Υ (cf. Chapter 4). The Planning-Agents α^ρ, any of these capabilities $c \in \mathcal{C}$ is associated within the plan ρ then can either be an Identified-Planning-Agent $\alpha^\rho{}_i \in \mathcal{A}^\rho_I \subset \mathcal{A}^\rho$ (and respective sets of Identified-Planning-Agent $\{\alpha^\rho_1, ..., \alpha^\rho_n\} \in \mathcal{A}^\rho_I$). Otherwise, the capability can address any of the Planning-Agent-Groups $\alpha^\rho_\forall, \alpha^\rho_\exists, \alpha^\rho_{\{\substack{\text{MIN}\\\text{MAX}}\}} \in \mathcal{A}^\rho_G \subset \mathcal{A}^\rho$, i.e., the All-Agent $\alpha^\rho_\forall$, the Any-Agent $\alpha^\rho_\exists$, or the Swarm-Agent $\alpha^\rho_{\{\substack{\text{MIN}\\\text{MAX}}\}}$, that do not address Identified-Planning-Agent directly (cf. Chapter 4). A capability-agent-association thus is defined as a surjective function $f_{ca}(c) : \mathcal{C}(\rho) \rightarrow \mathcal{A}^\rho(\rho)$ mapping each capability c named in ρ to a planning agent α^ρ named in ρ. This means that for enabling agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ to participate in the task allocation process by generating proposals subsequently, we need to map all capability-agent-association occurring in ρ into tasks $t \in \mathcal{T}^\rho$ we then include in a respective CfP. Each $t \in \mathcal{T}^\rho$ then contains information $\mathcal{C}_t$ concerning the required capabilities for assigning it. For generating this relevant set $\mathcal{C}_t$, we first need to define a task $t_{\alpha^\rho{}_i}$ for each different place-holding Identified-Planning-Agent $\alpha^\rho{}_i \in \mathcal{A}^\rho_I$ occurring in ρ and include $t_{\alpha^\rho{}_i}$ in a new set $\mathcal{T}^\rho_{\text{ID}} \subset \mathcal{T}^\rho$ of *identified tasks*, i.e.,

$$\mathcal{T}^\rho_{\text{ID}} := \{t_{\alpha^\rho_i} \mid \alpha^\rho_i \in \rho\} \tag{5.1}$$

For each of these tasks $t_{\alpha^\rho_i} \in \mathcal{T}^\rho_{\text{ID}}$ (which we call t^{ID} for short in the following), we can thus add all capabilities addressed to the respective Identified-Planning-Agent $\alpha^\rho{}_i$ named in ρ to a set of capabilities $\mathcal{C}_{t^{\text{ID}}}$ indicating which capabilities are required for assigning t^{ID}, i.e.,

$$\mathcal{C}_{t^{\text{ID}}} := \{c \mid f_{ca}(c) = \alpha^\rho{}_i\} \tag{5.2}$$

Second, we need to focus on the different types of unidentified Planning-Agent $\alpha^\rho \in \mathcal{A}_G^\rho$ named in ρ. Fortunately, we can handle the Any-Agent $\alpha_\exists^\rho$ and the Swarm-Agent $\alpha_{\{^{\text{MIN}}_{\text{MAX}}\}}^\rho$ (both from $\mathcal{A}_G^\rho$) equally as we can express $\alpha_\exists^\rho$ as an instance of $\alpha_{\{^{\text{MIN}}_{\text{MAX}}\}}^\rho$ with $\text{MIN} = \text{MAX} = 1$, i.e., $c_{\{1\}}^\rho$. For every occurrence of $\alpha_\exists^\rho$ or $\alpha_{\{^{\text{MIN}}_{\text{MAX}}\}}^\rho$ in ρ, we then need to define a further task t_{SW} which we collect in a set $\mathcal{T}_{\text{SW}}^\rho \subset \mathcal{T}^\rho$, i.e.,

$$\mathcal{T}_{\text{SW}}^\rho := \{t_{\text{SW}} \mid \alpha_{\{^{\text{MIN}}_{\text{MAX}}\}}^\rho \in \rho\} \tag{5.3}$$

Like for tasks $t^{\text{ID}} \in \mathcal{T}_{\text{ID}}^\rho$, we can define the set of required capabilities for each task $t_{\text{SW}} \in \mathcal{T}_{\text{SW}}^\rho$ by analyzing the respective capability-agent-associations made in ρ, i.e.,

$$\mathcal{C}_{t^{\text{SW}}} := \{c \mid f_{ca}(c) = \alpha_{\{^{\text{MIN}}_{\text{MAX}}\}}^\rho\} \tag{5.4}$$

Third, if there is any capability-agent-association named in ρ addressing the All-Agent $\alpha_\forall^\rho$, we do not need to introduce new tasks but modify those tasks t we have already included in $\mathcal{T}^\rho$, i.e., in one of its subsets $\mathcal{T}_{\text{ID}}^\rho$ and $\mathcal{T}_{\text{SW}}^\rho$. We thus extend the requirements of each of the existing tasks $t \in \mathcal{T}^\rho$ from $\mathcal{C}_t$ to $\mathcal{C}_t$' by adding those capabilities associated with any occurrence of $\alpha_\forall^\rho$, i.e.,

$$\forall t \in \mathcal{T}^\rho : \mathcal{C}_t{}' = \mathcal{C}_t \cup \{c \mid f_{ca}(c) = \alpha_\forall^\rho\} \tag{5.5}$$

That way, we can represent all requirements concerning capabilities the user-defined for any Planning-Agent $\alpha^\rho \in \mathcal{A}^\rho$ in the set of required task capabilities $\mathcal{C}_t{}'$ belonging to the respective tasks $t \in \mathcal{T}^\rho$.

5.4.2.2 Reacting to a (CfP) with a Distributed and Self-Aware Proposal Generation

As a response to a CfP, we require each agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ receiving that CfP to analyze the requirements defined by the tasks $t \in \mathcal{T}^\rho$ encoded in the respective sets of $\mathcal{C}_t$. To avoid confusion, we use $\mathcal{C}_t$ instead of $\mathcal{C}_t$' in this section again because an agent receiving the CfP does not know the pre-processing steps necessary for generating the final task requirements. Capabilities required included in any $\mathcal{C}_t$ for a task t can contain physical capabilities c_Υ^p of different types Υ and virtual capabilities c_Υ^v of different types Υ. For generating a valid response to the CfP, i.e., a proposal for the tasks $t \in \mathcal{T}^\rho$, each agent thus needs to evaluate its current set of provided capabilities $\mathcal{C}_\alpha$.

Referring to our Multipotent System reference architecture we introduce in Section 3.2, each $\alpha \in \mathcal{A}_{\mathcal{MS}}$ can only provide a specific physical capability c_Υ^p if all necessary hardware for c_Υ^p is connected. Thus, an agent α can execute, e.g., the capability $c_{\text{M-FIRE}}^p$ for detecting a fire when it has a respective hardware module connected (cf. the fire extinguisher module in Figure 5.5). If this is the case, we add the respective physical capability c_Υ^p to the set of provided capabilities $\mathcal{C}_\alpha \subset C$ of the agent α. In contrast to physical capabilities, with the concepts described on SEMANTIC HARDWARE LAYER (Section 3.2.8), we can also define virtual capabilities that do not directly require any hardware but specify their requirements concerning such within their parameters. Furthermore, also the *parametrization* of a virtual capability can require additional capabilities to be available for allowing an agent to execute it. This is the case for Collective Capabilities like we introduced them in Section 3.2.6. While we focus on the interesting part of Collective Capabilities, i.e., their execution, in Chapter 7 first,

we nevertheless need to ensure that their execution is possible after Ensemble Formation. A Collective Capability encapsulating the swarm behavior of PSO, e.g., typically is parametrized with a specific other capability representing the interest of the user. This can be, e.g., to find the source of a GAS_g (e.g., a fire) which requires the physical capabilities $c^p_{\text{GAS}_g}$ and $c^p_{\text{MV-VEL}}$ to be executable. Thus, a virtual capability $\mathcal{C}^v \in$ Collective Capabilities is only available to a α, if all capabilities included in the virtual capability's definition or its parameters are also available to the agent, i.e.,

$$\mathcal{C}^v \in \mathcal{C}_\alpha \Leftrightarrow \quad \forall c^p_\Upsilon \in \text{PAR}_{\mathcal{C}^v} \mid c^p \in \mathcal{C}_\alpha \tag{5.6}$$

Concerning each set of required capabilities $\mathcal{C}_t$ defined for each task $t \in \mathcal{T}^\rho$, the agent then can decide whether or not it could perform the task when selected in the task allocation afterward. The agent α thus encodes this decision in its proposal holding an entry for every task $t \in \mathcal{C}_t$, i.e., for $t^{\text{ID}} \in \mathcal{T}^\rho_{\text{ID}}$ and $t^{\text{SW}} \in \mathcal{T}^\rho_{\text{SW}}$.

$$\forall t \in \mathcal{T}^\rho : \quad \text{PRO}_\alpha(t) = \begin{cases} \text{TRUE} \Leftrightarrow & \mathcal{C}_t \subseteq \mathcal{C}_\alpha \\ \text{FALSE} & else \end{cases} \tag{5.7}$$

5.4.2.3 Task-Allocation Problem Based on Proposals

To form an ensemble $\mathcal{E}^\rho$ consisting of agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ that are collectively able to execute a certain plan ρ, we now need to formulate a Task Allocation Problem taking the proposals of these agents as input. To find a valid solution to the Task Allocation Problem, we then need to select such proposals that this selection collectively satisfies all requirements defined by any task $t \in \mathcal{T}^\rho$. Such a valid task allocation TA for ρ exists for the set of identified tasks $\mathcal{T}^\rho_{\text{ID}}$ if there is an injective function f_{TA} mapping each task $t^{\text{ID}} \in \mathcal{T}^\rho_{\text{ID}} \subset \mathcal{T}^\rho$ to a distinct agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ that proposed it can handle t^{ID}, i.e.,

$$\begin{aligned} \text{TA}(\mathcal{T}^\rho_{\text{ID}}) \Leftrightarrow \quad & \exists_{f_{\text{TA}}: T^{\text{ID}}_\rho \to \mathcal{A}_{\mathcal{MS}}} \forall_{t^{\text{ID}}_{j \neq k} \in T^{\text{ID}}_\rho} : \\ & f(t^{\text{ID}}_j) \neq f_{\text{TA}}(t^{\text{ID}}_k) \wedge \text{PRO}_{f_{\text{TA}}(t^{\text{ID}}_j)}(t^{\text{ID}}_j) = \text{TRUE} \wedge \text{PRO}_{f_{\text{TA}}(t^{\text{ID}}_k)}(t^{\text{ID}}_k) = \text{TRUE} \end{aligned} \tag{5.8}$$

While this solves the problem for all $t^{\text{ID}} \in \mathcal{T}^\rho_{\text{ID}}$ and generates a valid partial task allocation $\text{TA}(\mathcal{T}^\rho_{\text{ID}})$, we need to also include all tasks $t^{\text{SW}} \in \mathcal{T}^\rho_{\text{SW}}$ in our considerations for achieving a valid task allocation $\text{TA}(\mathcal{T}^\rho)$ for all tasks $t \in \mathcal{T}^\rho$ the plan ρ consists of (remember, $\mathcal{T}^\rho = \mathcal{T}^\rho_{\text{ID}} \cup \mathcal{T}^\rho_{\text{SW}}$). For this purpose, we need to also assure that the selection of proposals results in a valid assignment $\text{TA}(\mathcal{T}^\rho_{\text{SW}})$, i.e.,

$$\text{TA}(\mathcal{T}^\rho) = \quad \text{TA}(\mathcal{T}^\rho_{\text{ID}}) \wedge \text{TA}(\mathcal{T}^\rho_{\text{SW}}) \tag{5.9}$$

To suffice the requirements for $\text{TA}(\mathcal{T}^\rho_{\text{SW}})$, we need to assure to keep the number of agents $\mathcal{A}_{t^{\text{SW}}}$ that made proposals for every task $t^{\text{SW}} \in \mathcal{T}^\rho_{\text{SW}}$ we select in the defined bounds of $\text{MIN}_{t^{\text{SW}}}$ and $\text{MAX}_{t^{\text{SW}}}$ derived from the user's definition given with $\alpha^\rho_{\{\text{MIN}\atop\text{MAX}\}}$, i.e.,

$$\begin{aligned} \text{TA}(\mathcal{T}^\rho_{\text{SW}}) \quad \Leftrightarrow \quad & \forall t^{\text{SW}} \in \mathcal{T}^\rho_{\text{SW}} : \\ & \mathcal{A}_{t^{\text{SW}}} := \{\alpha \mid \text{PRO}_\alpha(t^{\text{SW}})\} \wedge \\ & \text{MIN}_{t^{\text{SW}}} < \mid \mathcal{A}_{t^{\text{SW}}} \mid < \text{MAX}_{t^{\text{SW}}} \end{aligned} \tag{5.10}$$

That way, we can guarantee that the selection of proposals we made fulfills all requirements defined in ρ concerning the number of required agents. Because all agents, we assign tasks

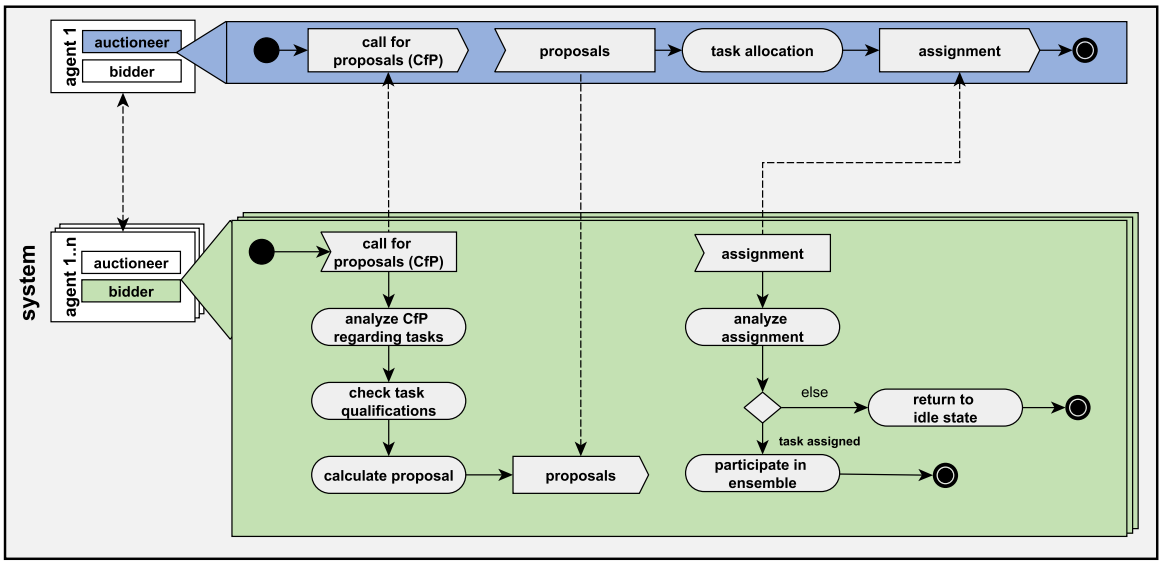


Figure 5.4: Algorithmic process for SELF-MADE solving the problem of Ensemble Formation as an instance of the Task Allocation Problem.

$t \in \mathcal{T}^{\rho}$ to, assured that they could satisfy the requirements defined in all tasks they agreed for in their proposals, we achieve a valid task allocation and thus form an appropriate ensemble with a mechanism implementing the given formalism.

5.4.3 Algorithmic Process of the Approach

Our algorithmic approach solves the problem of Ensemble Formation as an instance of the ST-MR-IA MRTA following the taxonomy of Gerkey and Matarić [2004]. We realize it following the general principle of the ContractNet protocol [Smith, 1980]. Figure 5.4 depicts the necessary process as an activity focusing on the interaction between agents participating in the process. At first, an auctioneer announces a new CfP when a new plan ρ becomes relevant for the Multipotent System $\mathcal{MS}$ after planning. In this CfP, the auctioneer encodes all requirements of ρ like we describe it in Section 5.4.2.1. Subsequently, the auctioneer sends this CfP to all agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ it can reach, increasing the range of possible participants and thereby the possibility of finding a valid allocation of tasks $t \in \mathcal{T}^{\rho}$ afterward. Agents receiving this CfP start analyzing the encoded requirements and decide on whether to participate in the process as bidders at all (they do not, if they are currently busy because of external factors like, e.g., reconfigurations of their hardware as we describe in Chapter 6) and then determine their qualification for each of the tasks enlisted in the CfP. According to their self-awareness based considerations, as we describe in Section 5.4.2.2, agents then generate proposals encoding the results of these considerations indicating whether they potentially can assign a task $t \in \mathcal{T}^{\rho}$ or not, following the formalization we introduce in Section 5.4.2.2. Agents send their proposals back to the auctioneer. After receiving the proposals, the auctioneer can start the task allocation process evaluating the information delivered in the proposals like we describe in Section 5.4.2.3. The result of this allocation is an assignment of tasks to agents. The auctioneer informs all agents whose proposals were considered during the task allocation about its decision. Agents receiving

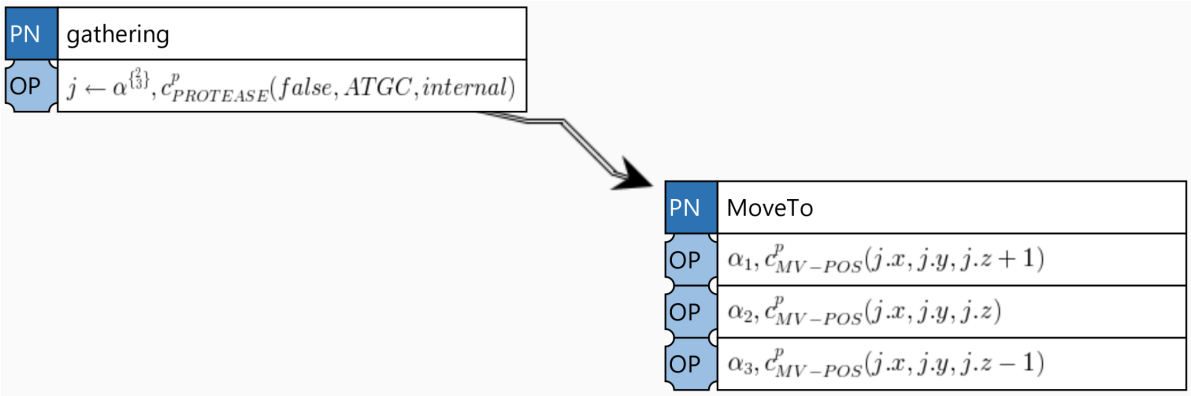


Figure 5.5: Exemplary plan addressing identified agents $\alpha^\rho_1, \alpha^\rho_2$, and α^ρ_3 in the Primitive-Node ⌈*PN:* moveTos⌋ and a Swarm-Agent $\alpha^\rho_{\{\text{MIN}\atop\text{MAX}\}}$ with MIN = 2 and MAX = 3 as a representative for an agent group in the Primitive-Node ⌈*PN* gathering⌋ (*moveTos* follows *gathering* by default without any further conditions, cf. Chapter 4). The ensemble consisting of three agents thus should first execute a gathering swarm behavior (encapsulated in the Collective Capability c^v_{PROTEASE}, cf. Section 7.5) before each of the three agents should move to a given position depending on the rendezvous position, while for the gathering, at least two agent should be able to participate. Figure exported from our prototypical reference implementation (cf. Section 3.3.5.1).

these decisions then analyze this information to determine whether they should perform one of the tasks they proposed to provide all required capability. If so, they participate in the ensemble $\mathcal{E}^\rho$ for the plan ρ. Otherwise, agents return in an idle state and wait for further instructions.

5.4.4 Modeling the Ensemble Formation Problem with MiniZinc

We formulate the process of proposal selection within SELF-MADE as a Task Allocation Problem using the formalism of CSOP (Section 5.4.1.1) and the MiniZinc constraint modeling language (Section 5.4.1.2). Depending on the actual problem we need to solve for a given plan ρ, we generate the respective MiniZinc model online. This enables us to only include all necessary details for each plan ρ. If e.g., the plan ρ does not include any references to agent groups but only includes identified agents, we avoid including constraints not necessary that would only increase the complexity of the model, resulting in higher calculation times.

We depict an exemplary version of one of these MiniZinc models encoding all necessary requirements for allocating and assigning the relevant tasks from an exemplary plan (cf. Figure 5.5) where $\mathcal{T}^\rho_{\text{ID}} \neq \emptyset$ and $\mathcal{T}^\rho_{\text{SW}} \neq \emptyset$ in Listing 5.2. As input parameters, we use

- the set of task $\mathcal{T}^\rho_{\text{ID}}$ and $\mathcal{T}^\rho_{\text{SW}}$, each represented with an index from the Integer domain (cf. *tasks_id* and *tasks_sw* in Line 3 to Line 6 of Listing 5.2).
- the set of agents participating in the allocation with proposals, each represented with an index from the Integer domain (cf. *agents* in Line 8 of Listing 5.2).
- the proposals of agents regarding identified tasks t^{ID}, represented as a 2-dimensional array from the Boolean domain, where each row represents the respective agent's decision concerning each task (cf. *proposals_id* in Line 9 to 11 of Listing 5.2).

- the proposals of agents regarding swarm tasks t^{SW}, represented as a 2-dimensional array from the Boolean domain, where each row represents the respective agent's decision concerning each task(cf. *proposals_sw* in Line 12 of Listing 5.2).
- the minimum and maximum number of required agents for each $t^{\text{SW}} \in \mathcal{T}^{\rho}_{\text{SW}}$ (cf. Line 4 and 6 of Listing 5.2).

The variables of the model are

- the allocation of identified tasks to agents (cf. *allocation_id* in Line 14 of Listing 5.2).
- the allocation of swarm tasks to agents (cf. *allocation_sw* in Line 15 of Listing 5.2).

The constraints of the model are

- the *all-differentconstraint* defined on the allocation, ensuring that all identified tasks are allocated to different agents in a valid allocation (cf. Line 17 of Listing 5.2).
- the *self-awareness constraint for* t^{ID}, ensuring each allocation of an identified task is only made if the agent proposed to provide all its requirements in its proposals (cf. Line 19 of Listing 5.2).
- the *self-awareness constraint for* t^{SW}, ensuring each allocation of a swarm task is only made if the agent proposed to provide all its requirements in its proposals (cf. Line 20 of Listing 5.2).
- the *swarm-member-bound constraint*, ensuring all swarm tasks $t^{\text{SW}} \in \mathcal{T}^{\rho}_{\text{SW}}$ are allocated to at least $\text{MIN}_{t^{\text{SW}}}$ and at maximum of $\text{MAX}_{t^{\text{SW}}}$ agents (cf. Line 23 and 24 of Listing 5.2).

We solve this model as a pure satisfaction problem (cf. *solve satisfy* in Line 26 of Listing 5.2) and are interessted in the allocation of identified tasks (cf. Line 28 of Listing 5.2) and swarm tasks (cf. Line 29 of Listing 5.2) to agents, that respects all constraints.

5.5 Evaluation

We evaluate the general feasibility of deploying and executing task allocation with SELF-MADE with the hardware we also successfully used with flying robots (cf. Section 3.4). In this study, we aim at achieving results in evaluating the scalability of our approach concerning the number of agents involved in task allocation and by evaluating our approach within simplified SCORe missions requiring multiple agents providing different capabilities to cooperate.

5.5.1 Feasibility and Scalability of Forming Flying Ensembles

To substantiate that Ensemble Formation is feasible with our approach also on hardware portable by an agent in a flying ensemble, we evaluated it on an Odroid XU4 [Hardkernel, 2018] with an Exynos5422 octa-core ARM processor and 2 GB of RAM single-board computer (Odroid) that is capable of running all necessary software, and that can be mounted, e.g., on UAV like we already used them in our evaluations in Section 3.4. In particular, we consider the ability to solving CSOP modeled with MiniZinc as we propose them in Section 5.4.4. Our evaluations, thus in general, address the question of whether it is feasible to use a state-of-the-art constraint solver on a single board computer for a practical problem.

```
1  include "alldifferent.mzn";
2
3  set of int: tasks_id = 1..3;
4  array[tasks_sw] of par int: tasks_sw_min = [2];
5  set of int: tasks_sw = 1..1;
6  array[tasks_sw] of par int: tasks_sw_max = [3];
7
8  set of int: agents = 1..3;
9  array[agents, tasks_id] of bool: proposals_id  =  [|true,true,true|
10                                                    true,true,true|
11                                                    true,true,true |];
12 array[agents, tasks_sw] of bool: proposals_sw = [|true|true|false|];
13
14 array[tasks_id] of var agents: allocation_id;
15 array[tasks_sw] of var set of agents: allocation_sw;
16
17 constraint alldifferent(allocation_id);
18
19 constraint forall(i in tasks_id)(proposals_id[allocation_id[i], i] = true);
20 constraint forall(i in tasks_sw) (forall(j in allocation_sw[i])
21     (proposals_sw[j, i] = true));
22
23 constraint forall(i in tasks_sw) (tasks_sw_min[i] <= card(allocation_sw[i]));
24 constraint forall(i in tasks_sw) (tasks_sw_max[i] >= card(allocation_sw[i]));
25
26 solve satisfy;
27
28 output[ "\(allocation_id)" ];
29 output[ "\(allocation_sw)" ];
30
```

Listing 5.2: MiniZinc Model generated for Figure 5.5

Figure 5.6: MiniZinc Model generated for allocating the exemplary plan ρ (cf. Figure 5.5) originating from a possible SCORe mission. We consider two types of tasks $t \in \mathcal{T}^{\rho}$ in this solution, enabling us to select proposals dedicated to adopting roles of concrete identified planning agents and that adopting roles of swarm members independently. The MiniZinc output for that concrete example thus is `[3, 2, 1][1..3]`. This indicates that α_1's proposal for t_3 was accepted, α_2's proposal for t_2, and α_3's proposal for t_1. Further, two of the three agents, i.e., α_1 and α_2, are also involved in the execution of the swarm behavior and α_3 is not (it cannot handle the task, cf. Line 12).

Evaluation Problem For our evaluations, we choose a well-defined *optimization* problem representing an even more complex variation of our model in Listing 5.2 (where we only require to solve a satisfaction problem). Because first evaluations indicated that we could solve the problem defined in Listing 5.2 in less than 200 milliseconds on average for plans involving up to 100 agents and 100 tasks, we chose to increase the problem's complexity instead of further increasing the problem's size for our evaluations in this chapter. In our evaluation model, we model an MRTA of the type ST-MR-IA, where we require n tasks of a plan to be assigned to n agents. For every agent/task pair, we additionally introduce a non-negative cost estimation if

the task is assigned to the agent. The objective of this problem was to minimize the worst costs any agent encounters. With this model, we thus already investigate future research directions to include some optimization criteria of relevance into our task allocation mechanism. For example, such can be, e.g., the assignment of agents to positions in a formation included in a plan, minimizing the farthest distance any agent has to move towards this position. That way, we find allocations that minimize the total duration to establish the formation. We can formalize this extended CSOP in the following way:

$$\begin{aligned} \underset{t_1,\ldots,t_n}{\text{minimize}} \quad & \max_{i\in\{1,\ldots n\}} \text{Cost}[i,t_i] \\ \text{subject to} \quad & i \neq j \rightarrow t_i \neq t_j \\ & t_i \in \{1,\ldots,n\} \end{aligned} \tag{5.11}$$

where t_i maps to the task agent i performs and $\text{Cost}[i,j]$ is the cost matrix (cf. the similarity compared to the model in Listing 5.2, where we let the agents make this decision only in general, i.e., according to whether they can assign the task or not, not respecting any quality measure).

Experiment Setup The problem's only parameters are n and Cost, which makes it attractive to generate random instances. Specifically, for a problem consisting of n agents and tasks, we generate random 3D coordinates in a rectangular cuboid with a width and length of 1000m and a height of 200m to represent a volume typical for a SCORe mission. We implemented a MiniZinc model that is optimized using Gecode.[1] Evaluation devices for our comparative studies were a desktop personal computer running a quad-core processor with 3.2 GHz and 16 GB of RAM, Ubuntu 15.10 (Desktop), and the aforementioned single-board computer, i.e., an Odroid with Ubuntu 14.04 as the operating system. We explored the solving behavior on identical problems for both hardware configurations with a timeout of 60 seconds. We decided that more extended enduring calculations would not be interesting in a real-world scenario involving flying ensembles. We let the problem size n range from 10 to 40, with 15 problems generated for each value of n. Quantities of interest we investigated in, were the time needed by the respective hardware to find an optimal solution (*duration*), the ratio of problems that were optimally solved (optimality), and the achieved objective (quality), i.e., the total costs of the solution. In Figure 5.7, we visualize the results of this comparative evaluation.

Comparing Duration In Figure 5.7, we see that the performance gap between the Desktop and the Odroid is rather low, i.e., there is only few difference in time needed for calculating solutions to the problem. For problems involving 10 to 15 agents, the average run-time ranged between 0.5 seconds and 1 second on the Odroid. On the Desktop, the average run-time ranged between 0.03 seconds and 0.15 seconds. For larger problem sizes, the average difference between the Odroid and the Desktop ranged up to few seconds only until both the Odroid and the Desktop converge to the timeout of 60 seconds when coming close to a problem size of $n = 40$.

Comparing Optimality Concerning the ratio of optimally solved problems, we found that the Odroid and the Desktop behaved very similarly. Only in five cases, the Odroid failed

[1]Source code at `https://git.io/vKeJx` ,MiniZinc on `http://www.minizinc.org/`, and Gecode on `http://www.gecode.org/`

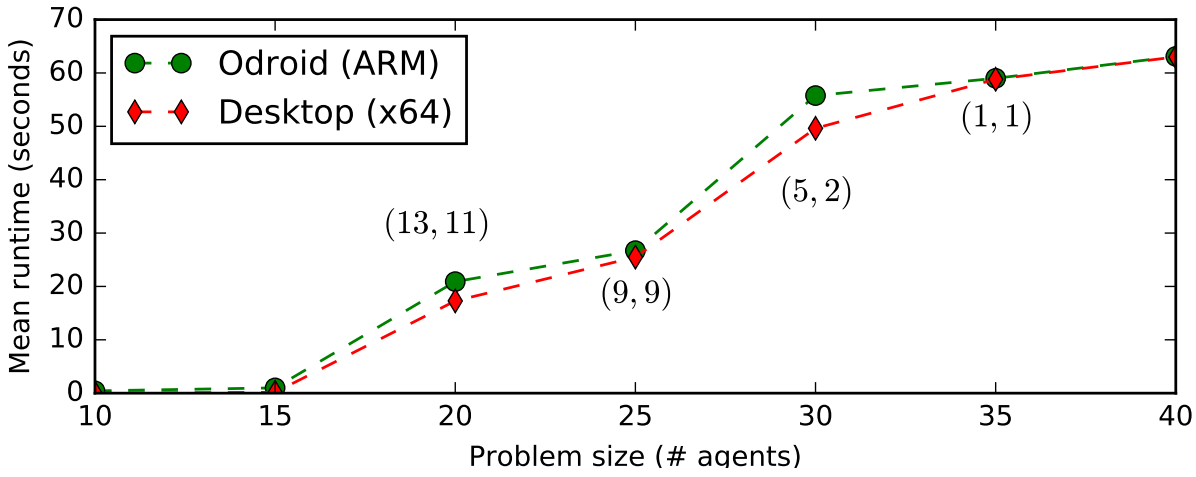
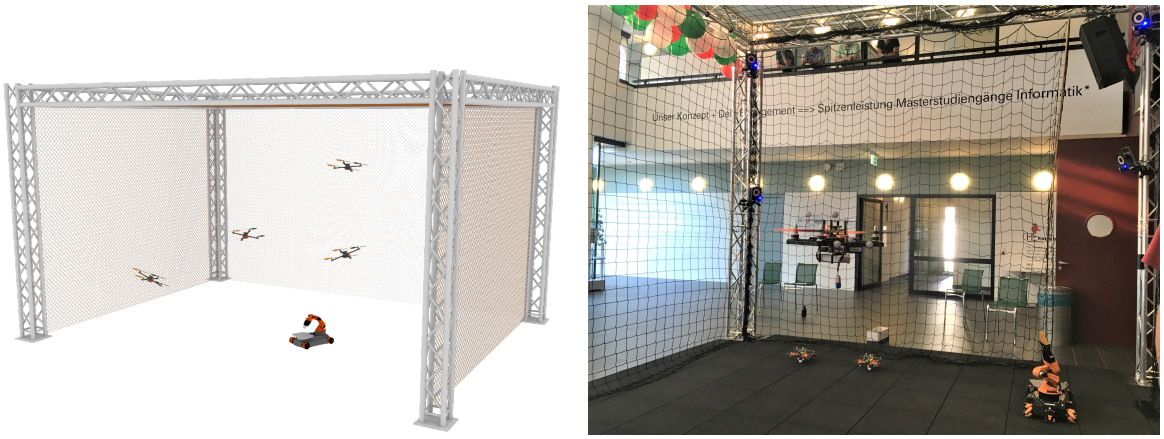


Figure 5.7: Run-time gap between an Desktop and an Odroid. Standard deviations are omitted for clarity. Values (x, y) indicate the number of problems (out of 15) the Desctop (x) and Odroid (y) could prove optimal (close). For $n \in \{10, 15\}$, all problems were closed, for $n = 40$, none was.

to prove optimality within the time limit of 60 seconds while the Desktop proved optimality within the time limit.

Comparing Solution Quality For comparing the quality of solutions found by the Odroid and the Desktop, we investigated the average solution quality for all problem instances in all problem sizes. We detect that the overhead concerning this quality regarding the best-found solution on the Odroid (after 60 seconds) compared to the Desktop was 2%. Further experiments suggest that, beyond $n = 40$, it becomes hard to prove optimality in the 60s, even for the Desktop.

Conclusion We can conclude that there is only a tiny performance gap when comparing the duration, optimality, and solution quality between a Desktop and an Odroid. Thus, we can assume that the Odroid, which we can mount on a flying agent within a flying ensemble, provides sufficient computational power for solving the Task Allocation Problem. We ground our finding on the results of our preliminary evaluations with the CSOP model written in MiniZinc, which illustrated the feasibility of finding solutions for Task Allocation Problems formulated as satisfaction problems even with very large problem sizes. Further, we found that when increasing the Task Allocation Problem's complexity by introducing quality measures for solutions, e.g., determined by agents and encoded in their proposals, we can even use single-board computers like the Odroid for problem sizes of up $n = 20$. With this problem size, we could not detect any significant difference between our evaluation devices. Neither the Odroid nor the Desktop we used in our comparative evaluation had trouble in proving optimality in this problem size. Thus, we can conclude that deploying and executing SELF-MADE in flying ensembles is feasible concerning calculating valid solutions to the Task Allocation Problem. Further, we can even use our approach when the need arises for also involving quality measures



(a) sketch of the arena (b) real arena

Figure 5.8: Flying arena used during the experiments. Figure 5.8a shows a schematic sketch, and Figure 5.8b the real arena.

in the process. In the next section, we demonstrate this feasibility in a proof-of-concept evaluation involving different real robot hardware.

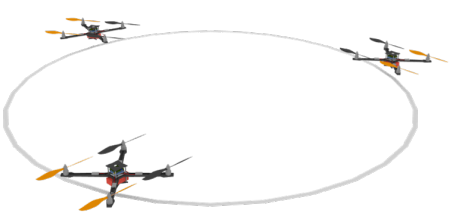
5.5.2 Ensemble Formation in a Real-World Setting

We developed a demonstrator to validate SELF-MADE which we use for task allocation in a real-world setting involving real robot hardware. In an indoor flying arena (cf. Figure 5.8), we installed for demonstration purposes during the 10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems 2016 (SASO2016) [University of Augsburg, 2016] which took place in our laboratory environments, we let ensembles work on user-defined plans cooperatively. Video materials showing our demonstration can be found on GitHub and YouTube[2] (video *SASO 2016 - Decentralized Coordination of Heterogeneous Ensembles Using Jadex*).

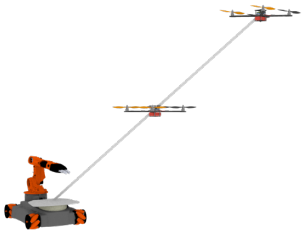
5.5.2.1 Experiment Setup and Test Bed

The experiment involved a heterogeneous ensemble consisting of flying agents as well as driving agents providing different capabilities concerning their movement range (cf. Figure 5.9c). Instead of using a graphical tool for designing SCORe missions we can derive situation-aware plans from (as we describe in Chapter 4), we developed another intuitive possibility for interacting with the ensemble sufficing the particular requirements of a public demonstration. With a pointing device (cf. Figure 5.9c), the user could create plans directly in-situ. Due to the limited spatial conditions we faced in our indoor environment, we reduced the flexibility in plan design to only involve movement capabilities $c^{p}_{\text{MV-POS}}$. With appropriate gestures using the pointing device, the user could determine the ensemble size, i.e., the number of agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ involved in the plan, and the pattern of a goal formation these agents should take cooperatively. Depending on the concrete positions the so-defined formation included, capabilities of different agents were required to allocate the tasks $t \in \mathcal{T}^{\rho}$. While tasks requiring agents

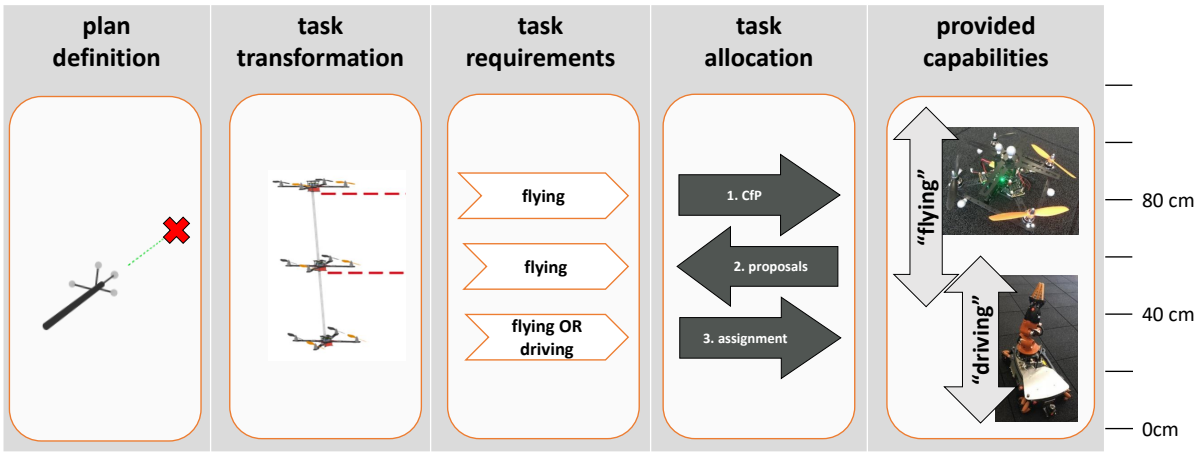
[2] `https://github.com/isse-augsburg/ensemble-programming` or `https://github.com/kosakoliver/ensemble-programming` or `https://www.youtube.com/user/ISSELabs`



(a) circle formation



(b) tower formation



(c) deducing task requirements from an exemplary user-defined plan

Figure 5.9: Schematic description of the experiment we performed for demonstrating the feasibility of deploying our mechanism for Ensemble Formation to real robots with an exemplary plan.

to execute $c^{p}_{\text{MV-POS}}$ parametrized with airborne positions could only be assigned to flying agents (starting at approximately 80 centimeters), tasks involving only near-ground positions could also be allocated by driving agents (down to approximately 40 centimeters). In contrast, tasks involving ground-based positions only could be assigned to driving robots (cf. Figure 5.9c). With our approach for Ensemble Formation, we deployed to each of the agents involved in the system then should find appropriate task allocations for the individual plan the user defined with the pointing device. We therefore implemented the approach presented in Section 5.4.3 using the MiniZinc model realizing the CSOP we present in Section 5.5.1.

5.5.2.2 Flying Arena and Hardware Prototypes

For setting up the indoor flying arena, we used the VICON external and optical tracking system [VICON, 2018], which we already used in our evaluations in Section 3.4.3. This tracking system can identify patterns of reflective spheres and determine their position with high accuracy. So we designed the pointing device that allowed the user to define plans with such spheres, enabling the system to automatically detect the concrete properties of the plan, i.e., the position of the plan's goal formation, the type of formation, and the number of agents involved in the formation

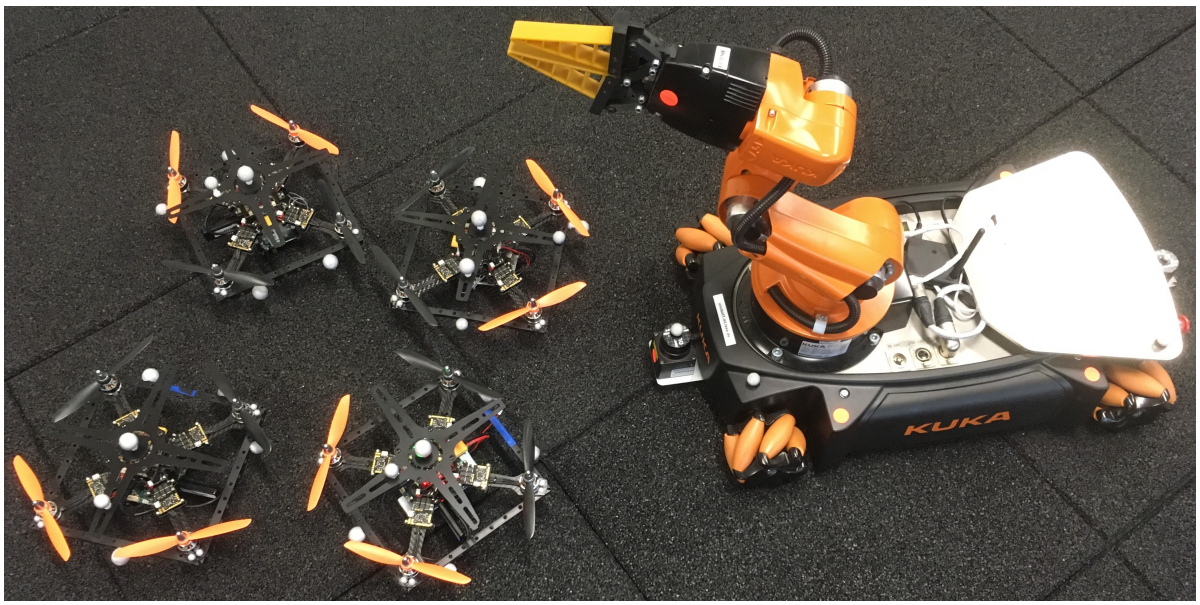


Figure 5.10: Flying and driving agents involved in the experiments used for demonstrating SELF-MADE in a simplified real world setting involving real hardware.

(cf. Figure 5.9). We further used the tracking system for determining the position of every agent in the experimental area, compensating for the missing Global Positioning System (GPS) typically available in outdoor experiments. That way, we enabled each agent to provide and execute the capabilities $c^p_{\text{MV-POS}}$ and $c^p_{\text{M-POS}}$ relevant for the possible plans. We used unique and asymmetrical patterns of these reflective spheres for each agent to distinguish different agents and the pointing device for associating tracked positions appropriately.

For realizing the flying agents (cf. Figure 5.10), we used UAV similar to those involved in the experiments in Section 3.4. Each flying agent we used carried an Odroid XU4 [Hardkernel, 2018] with an Exynos5422 octa-core ARM processor and 2 GB of RAM as we involved them in our feasibility and scalability evaluations in Section 5.5.1. Driving agents (cf. Figure 5.10) were based on the KUKA youBot mobile ground robot [KUKA, 2018], equipped with comparable computational power (Intel Atom Dual-Core CPU, 2 GB RAM). For carrying the spheres required for indoor tracking with the VICON system, we constructed specialized hardware prototypes for flying and driving agents, e.g., by expanding the frame of the UAV as illustrated in Figure 5.10.

5.5.2.3 Experiment Results

The system consisting of agents as we describe them in Section 5.5.2.2 was able to form appropriately composed ensembles automatically. We demonstrate this in multiple shows during the SASO2016 conference and in associated videos on GitHub and YouTube (video *SASO 2016 - Decentralized Coordination of Heterogeneous Ensembles Using Jadex*). Successfully executed experiments involved up to 5 different agents (four flying agents and one driving agent) that autonomously formed plan-specific ensembles. Plans differed in the number of agents involved (from 1 to 5 agents), positions in the experiment area (such positions that lay within a defined save space for avoiding harming visitors), and formations (tower formations and circle

formations). Calculation times for solving the underlying task allocation times were low for all experiments we executed, i.e., there were no latencies detectable for visitors of the demonstrations. The movement of agents for taking the respective formations further shows the benefits, including optimization criteria into the process of proposal selection as we investigated in in Section 5.5.1.

5.5.2.4 Conclusion

The experiments we performed on real hardware demonstrated the feasibility of the dynamic instruction of ensembles with user-defined plans. We showed that using SELF-MADE is also feasible in (simplified) real-world settings. Our experiments showcase that agents can evaluate their provided capabilities using appropriate self-awareness functionalities for providing appropriate proposals depending on the individual plans' requirements. Moreover, with our experiments, we further provide a proof-of-concept that hardware we can use for constructing flying agents can provide sufficient computational power for processing such proposals to find valid and even optimal task allocations.

5.6 Future Research Directions

Our findings in Section 5.5 show that for solving the CSOP as a satisfaction problem only, i.e., ensuring all requirements of a plan ρ hold with the selected proposals representing the task allocation for ρ, we this deployment is feasible also when scaling the problem size to large size systems involving up to 100 agents. Therefore, we already investigated increasing the problem complexity by integrating optimization criteria when selecting proposals. Results from that evaluations are also very promising, indicating that for middle-sized systems deploying such selection mechanisms is feasible also for flying ensembles' hardware. Future research can investigate this finding by investigating possible optimization criteria a user can define for a plan. Finding such valid plans involving multiple capabilities in their tasks' requirements turned out to be non-trivial. By introducing quality measures for capabilities, we assume that the complexity of selecting the best proposals for the tasks in a plan increases immensely. When introducing such, future research needs to focus on measures making different capabilities comparable in situations where some agents can provide high quality in executing capability A while only a reduced quality in executing capability B and other agents might provide those capabilities with an inverse quality. With an increasing set of possible capabilities involved in a plan by the user, making individual proposals comparable by simple measures like introducing internal weights might not be sufficient. Instead, the approach of Constraint Preferences for Soft Constraints [Schiendorfer et al., 2013] proposes to deliver appropriate measures for achieving good selections of proposals while also being able to handle situations involving incomparable options. With the implementation in MiniBrass [Schiendorfer et al., 2018] being an extension to MiniZinc, we technically already opened the door for such future research.

Chapter Summary and Outlook

In this chapter, we handled the problem of Ensemble Formation relevant for creating ensembles within Multipotent System that are capable of working on plans derived from SCORe missions. We proposed to address this problem with a Distributed, Self-Aware Market-Based Mechanism for Ensemble Formation (SELF-MADE). To achieve this, we mapped the act of Ensemble Formation to the underlying problem of task allocation that is known to be solvable by coalition formation [Shehory and Kraus, 1998]. This approach can deal with the special properties of Multipotent System, i.e., distributed deployment, possible high scale of participating agents, and changing capabilities of agents. We formalized the requirements for a plan for which the system should form an ensemble according to the user's definitions in individual tasks each. For finding solutions to the problem, we express these requirements in a Constraint Satisfaction (and Optimization) Problem (CSOP) we model using the MiniZinc constraint modeling language. For achieving solutions to the Task Allocation Problem, we implemented SELF-MADE by adapting the well known ContractNet protocol [Smith, 1980], consisting of a phase for proposal generation as a response to an Call for Proposals (CfP), and a selection phase for determining which proposals to accept for fulfilling all requirements of a plan. We demonstrated the feasibility of deploying that solution to flying ensembles by evaluating its scalability using real hardware and showcasing its application with a real flying ensemble in multiple indoor laboratory experiments.

In this chapter, we assumed that for each possible plan we could generate from a SCORe mission, we have enough sufficiently equipped agents available to fulfill all requirements of the plan. Unfortunately, we can not maintain this assumption for flying ensembles in general. When configuring the hardware composition of flying agents in Multipotent Systems, we need to consider their maximum additional payloads. Thus, we can neither guarantee that all agents can provide all required capabilities of all tasks in a plan at every time nor have at least as many sufficiently equipped agents available as we require to form a respective ensemble for a plan. To overcome this issue, we propose a solution to resolving situations where such ensembles can not be formed with the current configuration of the Multipotent System.

Chapter 6

A Self-Organization Mechanism for Physical Reconfiguration

Summary. In the last chapter, we assumed that the current configurations of agents were sufficient to find an ensemble for a specific plan in a given Multipotent System. Now we consider the situation that plans can, in principle, be satisfied by the Multipotent System but physical reconfigurations are necessary. For such situations, we require the Multipotent System to find one of these valid new physical configurations concerning the physical hardware configurations of robots controlled by the agents in the Multipotent System which defines their respective set of provided capabilities. Then, the Multipotent System again can autonomously form an ensemble for handling a plan the Multipotent System could not handle before. We find that we can formulate searching for that configuration as an instance of the Resource Allocation Problem (RAP). For solving the RAP in general, we propose a heuristic and market-based approach that exploits the possibility of domain-specific problem decomposition. Our approach finds solutions to the corresponding Constraint Satisfaction (and Optimization) Problem (CSOP) in a distributed fashion, based on the paradigm of self-organization. We empirically show that this distributed approach significantly outperforms a centralized one in a broad set of experiments. This finding holds when increasing the problem size regarding the number of agents, the number of tasks in the plan, and the number of capabilities relevant for the respective case study (i.e., SCORe mission) the plan originates from. We further empirically show within a wide range of experiments that the distributed solution, in general, provides the same solution quality as the centralized version. By integrating that solution into the layers of our Multipotent System reference architecture as we describe it in Chapter 3, we enable the core feature of our approach: Agents in Multipotent System no longer need to be individualized for different use-cases and tasks. Instead, we enable Multipotent Systems to adapt to changing requirements as needed by the user, i.e., as defined in the respective plans relevant in different situations.

Publication. Contents of this chapter have been published in [Hanke et al., 2018; Kosak et al., 2018].

6.1 Physical Reconfiguration in Multipotent Systems

In Chapters 1 and 2, we elaborated that flying ensembles can provide helpful support in versatile forms of applications that offer the potential to improve human life in a multitude of different areas. We further found in the respective literature research in Chapters 1 and 2 that current applications already offer a vast set of heterogeneous capabilities for this wide range of use cases. We analyzed the drawback of this development to be that each of these applications relies on specifically constructed agents and hardware tailored to the specific use case. To overcome this need for specialization, we introduced the approach of Multipotent Systems in Chapter 3. The core functionality of Multipotent System is that we do not need to couple capabilities $c \in \mathcal{C}$ and agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ in a fixed way. Instead, we can adapt the configuration of agents $\mathcal{SDH}_\alpha$ concerning their hardware and thereby modify the resulting set of provided capabilities $\mathcal{C}_\alpha$ of agents as required by the application and use-case.

In this chapter, we now focus on enabling this essential feature. We propose a mechanism for re-allocating hardware we assume to be available in the form of SDH and thereby modify the capabilities of agents as required. We define the underlying problem as an instance of the Resource Allocation Problem (RAP). In this RAP, *resources* are SDH that can provide certain capabilities to agents $\alpha \in \mathcal{MS}$ they are connected to. *Resource holders* then are those agents $\alpha \in \mathcal{MS}$ whose set of capabilities $\mathcal{C}_\alpha$ we can change when re-allocating resources.

6.1.1 Embedding in the Case Study of Dealing with Gas Accidents

We do not restrict ourselves to one use case but develop an approach to resource allocation that can be deployed in various case studies. Nevertheless, in this chapter, we evaluate our work on a scenario from our case study of *Dealing with Gas Accidents* we introduced in Section 2.5. Therein, firefighters use flying ensembles to support the handling of chemical gas accidents autonomously. With the methodologies we introduced in Chapter 4, we can define a respective SCORe mission HTN_{GAS} for that case study. Like we described in Section 2.5, we may require a broad set of different capabilities for handling possible plans ρ resulting from HTN_{GAS}. Figure 6.1 depicts how a concrete instance of a Multipotent System, i.e., $\mathcal{MS}_{\text{GAS}}$, can handle a SCORe mission like we introduced it in Figure 2.7. For $\mathcal{MS}_{\text{GAS}}$ with $\mathcal{A}_{\text{GAS}}$ consisting of 10 agents, i.e., $|\mathcal{MS}_{\text{GAS}}| = 10$, we show possible assignments of tasks from the respective plans in Figure 6.1. With our reduced set of different agents $\alpha \in \mathcal{A}_{\text{GAS}}$, we can assume that we require to let $\mathcal{MS}_{\text{GAS}}$ adapt the $\mathcal{SDH}_\alpha$ of one or multiple $\alpha \in \mathcal{A}_{\text{GAS}}$ at positions R_1, R_2, and R_3 (the points in the course of a SCORe mission where the phase changes). Thus, at R_1, R_2, and R_3, we require to let $\mathcal{MS}_{\text{GAS}}$ solve the RAP by generating a new allocation of SDH $\in \mathcal{SDH}_{\text{GAS}}$ to $\alpha \in \mathcal{A}_{\text{GAS}}$. In comparison to these 10 agents, a conventional heterogeneous system with fixed hardware-agent configurations would require a much larger set of $\mathcal{A}_{\text{GAS}}$ for achieving the same outcome:

- The Search phase of the SCORe mission requires 10 agents configured with an SDH combination of $\{\text{SDH}_{\text{ALL}}\}$ for providing all required capabilities formalized in a respectively generated plan $\rho_{\text{GAS-S}}$, i.e., we require *10 agents* $:= 10 \cdot |\{SDH_{ALL}\}|$)
- The Continuously Observe phase of the SCORe mission requires three agents configured with an SDH combination of $\{\text{SDH}_{\text{X}}, \text{SDH}_{\text{WE}}\}$, two agents with $\{\text{SDH}_{\text{ALL}}\}$, and five agents with $\{\text{SDH}_{\text{X}}, \text{SDH}_{\text{CA}}\}$ for providing all required capabilities formalized in a respectively generated plan $\rho_{\text{GAS-CO}}$. We can reuse 2 already correctly configured agents with an SDH

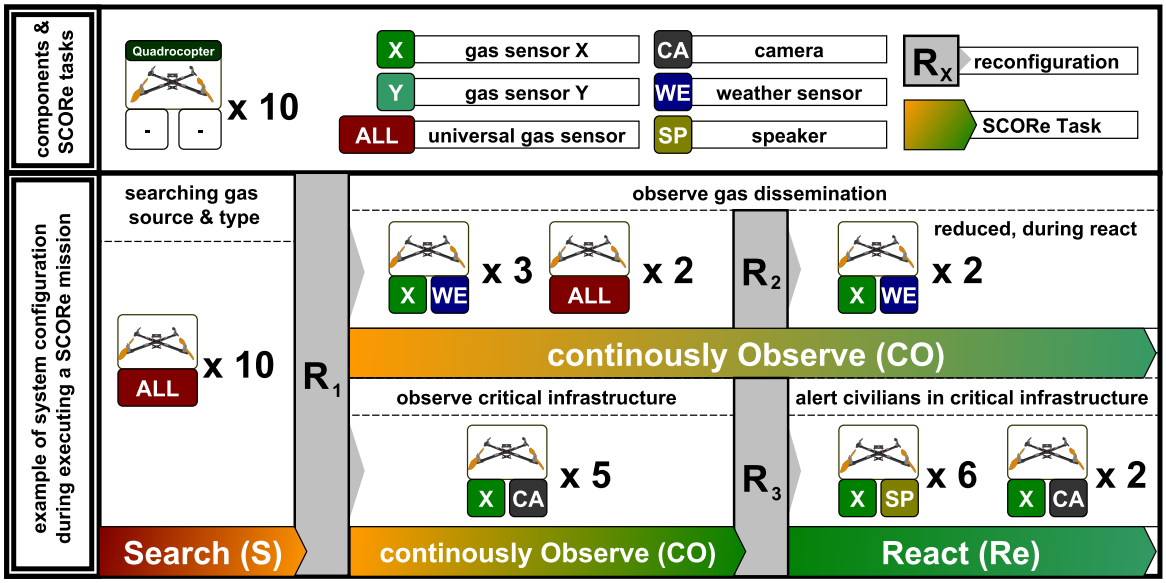


Figure 6.1: Example of a SCORe mission for the case study of *Dealing with Gas Accidents*, focusing on the configuration of agents $\alpha \in \mathcal{A}_{\text{GAS}}$. The legend (top part) shows the concrete instantiation of a $\mathcal{MS}_{\text{GAS}}$ with its available agents $\mathcal{A}_{\text{GAS}}$ and hardware $\mathcal{SDH}_{\text{GAS}}$. The bottom part shows changes in the configuration of $\mathcal{MS}_{\text{GAS}}$ (i.e., changes in the $\mathcal{SDH}_\alpha$ for $\alpha \in \mathcal{A}_{\text{GAS}}$) in the course of the SCORe mission. R_1, R_2, R_3 indicate situations where adaptations of this configuration, i.e., resource re-allocations, may happen within $\mathcal{MS}_{\text{GAS}}$.

combination of $\{\text{SDH}_{\text{ALL}}\}$ from the Search part, i.e., we require *8 additional agents* $:= 3 \cdot |\{SDH_X, SDH_{WE}\}| + 5 \cdot |\{SDH_X, SDH_{CA}\}|$).

- The React phase of the SCORe mission requires six agents configured with an SDH combination of $\{\text{SDH}_{\text{X}}, \text{SDH}_{\text{SP}}\}$ and two agents with an SDH combination of $\{\text{SDH}_{\text{X}}, \text{SDH}_{\text{CA}}\}$ for providing all required capabilities formalized in a respectively generated plan $\rho_{\text{GAS-RE}}$. Again, we can reuse 2 already correctly configured agents from the React part, i.e., we require *6 additional agents* $:= 6 \cdot |\{SDH_X, SDH_{SP}\}|$)

Thus, we would require 24 agents in a conventional heterogeneous system while we require only 10 in a Multipotent System. To identify the set of SDH needed for providing certain capabilities, agents in Multipotent System are able to query a *common knowledge data base* defining the relation between hardware types and capabilities (cf. Section 3.2.8). Given, e.g., the set of $\mathcal{SDH}_\alpha = \{\text{SDH}_{\text{X}}, \text{SDH}_{\text{WE}}\}$, agent α gains the physical capabilities $c^p_{\text{M-GASX}}$ (from SDH_{X}), $c^p_{\text{M-WIND}}$ (from SDH_{WE}), and a combined virtual capability $c^p_{\text{M-DISTRIBUTION}} := \{c^p_{\text{M-GASX}} \bigstar c^p_{\text{M-WIND}}\}$ for estimating the gas cloud distribution (that uses both $\text{SDH}_{\text{X}}$and SDH_{WE} to predict the future gas dissemination).

6.1.2 Assumptions and Structure in this Chapter

In our studies in this chapter, we neglect the quality of S&A encapsulated in the respective SDH (e.g., the measuring frequency of a sensor) and focus on their types only. Nevertheless, in all of the following definitions and algorithms, hardware module quality can be easily considered.

We further assume that every agent is a flying agent. This means that every agent has one fixed part in its hardware configuration that provides it with moving capabilities as long as a fixed maximal additional payload is not exceeded. In a Multipotent System $\mathcal{MS}$, this would be a fixed element in $\mathcal{SDH}_\alpha$ for each agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ providing $c^p_{\text{MV-POS}}$ and $c^p_{\text{MV-VEL}}$ with a respective negative weight (cf. Section 3.2). We also assume that SDH physically and technically can be plugged in each other infinitely (e.g., as daisy chain [Andreas et al., 2005]) with the technologies developed in the project COMBO[1]. We use our Jadex-based prototypical implementation of our Multipotent System reference architecture as we describe it in Section 3.3 to implement the reconfiguration.

We organize the rest of this chapter as follows: In Section 6.2, we describe the RAP to be solved. We then investigate related work in Section 6.3, concerning the RAP in general and how it currently is applied to MAS/MRS. Subsequently, we propose our algorithmic approach for solving the RAP in Section 6.4 and present our results derived through empirical evaluations in Section 6.5.

6.2 Physical Reconfiguration as Resource Allocation Problem (RAP)

The problem we describe in Section 6.1 is no specific problem of Multipotent System. Instead, system developers may also face similar problems in very different situations. To illustrate this generality, we first analyze the specific problem we need to solve in Multipotent System, which integrates a Task Allocation Problem with a RAP. We then propose a general procedure for solving that integrated problem. Further, we formally specify the RAP and give a possible decomposition of that problem we can apply in MAS/MRS generally.

6.2.1 Defining the RAP in Multipotent Systems

Regarding the situation motivated in Section 6.1, the central problem in Multipotent System we need to solve is that of forming an ensemble for handling a plan as we describe it in Chapter 5. The concrete problem instance then is defined by the number of tasks contained in such a plan, including their requirements concerning capabilities agents need to provide when allocating the tasks. In our approach of Multipotent System, these requirements stem from the SCORe mission's definition performed by the Multipotent System's user. In these plans, we typically require multiple agents to cooperate. Further, we consider agents in Multipotent System to execute at most one task at a time. Thus, for working through a plan ρ, we require the Multipotent System to find a solution to the Task Allocation Problem where each task of ρ is allocated to one agent capable of handling it exclusively, i.e., an agent providing all capabilities that are required for solving the task. The group of these agents then forms the respective ensemble $\mathcal{E}^\rho$ able to handle ρ. This mechanism relies on having enough agents available to provide the required capabilities. As a consequence, the process of Ensemble Formation fails if no such coalition can be found. Then, the Multipotent System finds itself in a situation where the Task Allocation Problem is unsolvable, at least for the system's current configuration. Instead of adding more appropriately configured agents to the system (a typical solution how other approaches like that of Marconi et al. [2012]; Dias et al. [2006]; Korsah et al.

[1]COMBO - *Kombination von Planung, Selbst-Organisation und Rekonfiguration in einem Roboterensemble zur Ausführung von SCORe Missionen* - grant number 402956354.

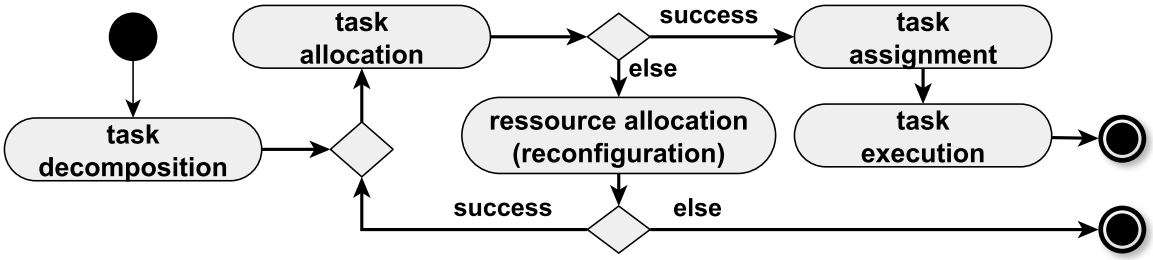


Figure 6.2: General procedure for solving the integrated problem of task an resource allocation in a MAS that provides the possibility of reconfigurating the agents' capabilities. If tasks cannot be allocated, a new resource allocation is calculated and the task allocation is restarted.

[2013] handle such situations), our approach in Multipotent System allows for reconfiguring the agents that are already participating in the system in terms of their provided capabilities. To comprehend how this general solution can again be integrated into the Multipotent System reference architecture, we refer the reader to Section 3.2, where we analyzed this in detail in our descriptions for the ENSEMBLE LAYER, the AGENT LAYER, and the SEMANTIC HARDWARE LAYER.

6.2.2 Proposing a Generalized Solution for Solving the RAP

As we analyzed in Chapter 5, we can generalize Ensemble Formation to be an instance of the Task Allocation Problem as defined by [Shehory and Kraus, 1998]. Concerning the taxonomy introduced by Gerkey and Matarić [2004] for classifying *Multi-Robot Task Allocation Problems (MRTA)*, we further identified it to be a *Multi-Robot Single-Task* allocation problem where tasks are instantly assigned (*ST-SR-IA*). As Shehory and Kraus [1998] proposes, we can find a solution to this problem by coalition formation. In situations where the system's configuration does not comply with the requirements of the tasks, i.e., no coalition can be formed within the system, this Task Allocation Problem gets augmented by a *Multi-Agent Resource Allocation (MARA)* [Chevaleyre et al., 2006]. By their very nature, resources (i.e., hardware modules) in that MARA are discrete, indivisible, not sharable, static, and multi-unit, when we classify them according to the definitions of Chevaleyre et al. [2006]. As we can see in Figure 6.2, we can resolve the situation if we enable the system to reallocate resources to new resource holders (i.e., the agents). By considering the respective unsolvable tasks' required capabilities as requirements for the RAP, we need to find a consistent system configuration for successfully finishing a subsequently re-started task allocation. Separating the process into an independent task and resource allocation reduces the complexity of the problem in general. This eases finding task allocations when the system is already configured appropriately (cf. Chapter 5). Further, separating the problem into a task and a resource allocation problem reduces the problem size in critical situations where we need the time we spend on calculating new configurations for our system to be minimal. By combining task and resource allocation as we propose in Figure 6.2, a valid task assignment can be found in every case where there is at least one configuration of the system sufficing the task requirements introduced by the user, i.e., there is at least one coalition with one configuration of agents that can handle the

tasks. The solution to the RAP we propose in this chapter thus is not restricted to be applied in Multipotent System only. Instead, it can be applied to any system

- where requirements can be formulated as one or multiple tasks,
- that faces changes in these requirements during run-time,
- that consists of one or multiple agents that can assign tasks if they fulfill all their requirements with the capabilities they provide,
- whose agents cannot fulfill all possible requirements at the same time,
- that offers the possibility to adapt the resource allocation of agents (e.g., concerning their connected hardware), and
- where changes in that resource allocation of agents influence the agents' capabilities and thereby change their qualification for tasks.

Thus, in the following, we do not restrict the statements to agents and hardware designed according to our Multipotent System reference architecture but refer to all systems that provide the properties mentioned above.

6.2.3 A Formal Specification of the RAP

We can formulate the resulting RAP as an instance of a Constraint Satisfaction (and Optimization) Problem (CSOP) (cf. Section 5.4.1.1) that minimizes the required amount of reconfigurations r_a concerning hardware modules, aggregated over all participating agents $\alpha \in \mathcal{A}$. This minimization problem is subject to multiple constraints regarding participating agents and relevant tasks t that define the requirements of the resource allocation problem. More precisely, these requirements are defined by the set of capabilities $\mathcal{C}_t$ each task $t \in \mathcal{T}$ requires from an agent to make it capable of handling the task.[2]

The solution of the CSOP then shall guarantee that for each task, there is at least one agent that is capable of handling it as the agent provides all capabilities C_α that are required by the task ($C_t \subseteq C_\alpha$). Because this allocation is still subject to the ST-SR, we define it by an injective function $f : \mathcal{T} \rightarrowtail \mathcal{A}$ (*task-covering-contraint*, cf. Equation (6.3)) that maps a unique agent $a \in A$ to every task $t \in \mathcal{T}$, ensuring that $C_t \subseteq C_{f(t)}$ is true for at least one agent. While f can also be used by the task allocation afterward (if enriched with other relevant constraints, e.g., for allocating tasks to the fastest or the nearest agent), we want to abstract from defining a concrete function during the resource allocation for allowing more possible solutions and thus more flexibility. The set of capabilities C_α an agent provides results from the set of hardware allocated to the agent H_α^{post} during the resource allocation process.[3] H_α^{post} is the set of hardware allocated to α by the solution of the resource re-allocation, being the set of decision variables for the CSOP. The relation between allocated hardware and provided capabilities is defined by a *common knowledge data base* that, queried with a particular set of hardware types, delivers the resulting capabilities as described in Section 6.1.[4] We can see this knowledge database as some non-complicated look-up table, not involving any sophisticated reasoning mechanism or else. As every hardware has a defined weight $\textsc{weight}(h)$ and each agent has a maximum payload $\textsc{pay}(\alpha)$, i.e., the maximum aggregated weight an agent is able to move with (in this chapter we assume this to be an inherent capability of each agent,

[2] in a Multipotent System, these requirements are defined by $\mathcal{T}^\rho$ for a specific plan ρ

[3] in a Multipotent System, H_α is defined by $\mathcal{SDH}_\alpha$

[4] in Multipotent System, we can also involve virtual capabilities that way

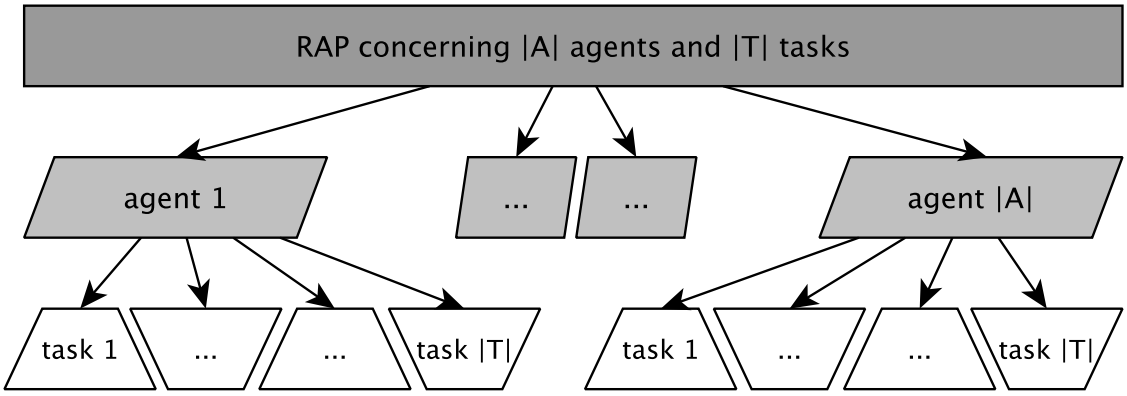


Figure 6.3: Two step decomposition of the RAP. While the centralized problem definition from Section 6.2.3 has to handle all $|\mathcal{A}|$ agents and $|\mathcal{T}|$ a decomposed partial problem only has to regard one agent and one task at a time.

cf. Section 6.1.2[5]), the allocation also has to take care of not overloading an agent (*payload-constraint*, cf. Equation (6.1)). Of course, each hardware module can only be allocated to one agent at a time (*hardware-constraint*, cf. (6.2)). We define the CSOP as follows:

$$
\begin{aligned}
\text{min.} \quad & \textstyle\sum_{\alpha \in \mathcal{A}} r_\alpha \\
\text{s. t.} \quad & \forall \alpha \in \mathcal{A} : \text{PAY}(\alpha) \geq \textstyle\sum_{h_\alpha \in H_\alpha^{post}} \text{WEIGHT}(h_\alpha), && (6.1)\\
& \forall \alpha_1 \neq \alpha_2 \in \mathcal{A} : H_{\alpha_1}^{post} \cap H_{\alpha_2}^{post} = \emptyset, && (6.2)\\
& \exists f : \mathcal{T} \rightarrowtail \mathcal{A}, \forall t \in \mathcal{T}, \mathcal{C}_t \subseteq \mathcal{C}_{f(t)} && (6.3)\\
\text{with} \quad & r_\alpha := \left| H_\alpha^{pre} \ \Delta \ H_\alpha^{post} \right|, && (6.4)\\
& \mathcal{C}_\alpha \subseteq \mathcal{C} := query\_database(H_\alpha^{post}), \\
& \mathcal{C}_t \subseteq \mathcal{C}, H_\alpha^{pre}, H_\alpha^{post} \subseteq H
\end{aligned}
$$

where $\mathcal{A}$ is the set of all available agents, $\mathcal{T}$ the set of all relevant tasks, H the set of all hardware modules, $\mathcal{C}$ the set of all capabilities, H_α^{pre} is the set of hardware allocated to agent $\alpha \in \mathcal{A}$ previous to an resource re-allocation.[6] In Equation (6.4) we use the symmetric difference $H_\alpha^{pre} \Delta H_\alpha^{post} := (H_\alpha^{pre} \setminus H_a^{post}) \cup (H_\alpha^{post} \setminus H_\alpha^{pre})$ to identify this set of changed hardware.

6.2.4 A Domain-Specific Decomposition of the RAP

When analyzing the complexity of the CSOP in Section 6.2.3, it is obvious that its traceability is only given for small sizes of H, $\mathcal{A}$, and $\mathcal{T}$. As for every agent there has to be decided which of the available hardware modules should be allocated to allow a valid task allocation, the number of solutions of the RAP is in $O(|\mathcal{A}|^{|H|} \cdot |\mathcal{T}|^{|\mathcal{A}|})$. In other words, for every possible

[5] We can also derive it by respecting such SDH enabling an agent with movement capabilities with negative weight (cf. Section 3.2.2) whose negated value we can use as $\text{PAY}(\alpha)$.

[6] in a Multipotent System, H_α^{pre} would be α's set of SDH that was relevant for a previous plan, e.g., $\text{SDHs}_\alpha^{\text{GAS-S}}$ when switching from $\rho_{\text{GAS-S}}$ to $\rho_{\text{GAS-CO}}$

resource allocation (i.e., $|\mathcal{A}|^{|H|}$ combinations [Chevaleyre et al., 2006]) we need to validate that a task allocation is possible (i.e., $|\mathcal{T}|^{|\mathcal{A}|}$ combinations [Gerkey and Matarić, 2004]). Considering the worst case $|\mathcal{T}| = |\mathcal{A}|$, the problem thus is in $O(|\mathcal{A}|^{|H|+|\mathcal{A}|})$. To alleviate this complexity, we propose to decompose the RAP, generate partial solutions, and aggregate them to achieve a solution approximately similar to the central solution concerning the optimization criteria, i.e., minimizing hardware reconfigurations (this heuristic already proved to be very efficient for resource allocation in the domain of virtual power plants we studied earlier [Kosak et al., 2015]). Due to the global component of the RAP (cf. Equation (6.2)) we prefer this Contract-Net inspired solution [Smith, 1980] over specifying our problem as a distributed constraint optimization problem (DCOP) [Fioretto et al., 2016]. In doing so, we use a problem-specific decomposition and avoid messaging overhead. The decomposition can be achieved by dividing the calculation of H_α^{post} for all $\alpha \in \mathcal{A}$ into partial problems concerning only a single agent $\alpha \in \mathcal{A}$, each. The resulting $|\mathcal{A}|$ partial problems are individualized for each agent, thus, instead of regarding all other agents when minimizing the amount of hardware reconfigurations as it is done in the centralized CSOP (cf. Equation (6.1)), each agent α only regards its own configuration, e.g., minimizes r_α and not $\sum_{\alpha \in A} r_\alpha$. Consequently, Equation (6.2) is neglected here but respected in the following solution aggregation. Resulting from the definition of the Task Allocation Problem as ST-SR (cf. Section 6.2.3), the problem size of the RAP is further reduced in terms of the tasks that are regarded while calculating a partial solution. More precisely, for solving one partial problem, only one specific task $\tau \in \mathcal{T}$ has to be regarded at a time. To cover all tasks of the original problem, the partial problem has to be solved for every task (i.e., $|\mathcal{T}|$ times) by each agent α consequently (cf. Figure 6.3). In the tradition of auction mechanisms, which are the most common market-based approaches [Dias et al., 2006], we call a solution to one partial problem *proposal*, i.e., the solution for the partial problem concerning agent α and task τ is named $H_\alpha^{pro}(\tau)$. This partial RAP again is encapsulated in a CSOP. For generating a valid proposal, the calculated partial solution $H_\alpha^{pro}(\tau)$ has to assert that the proposed resource allocation results in all relevant capabilities needed for the defined task τ (*capability-constraint*, cf. Equation (6.5)), when queried within the data base while respecting agent α's *payload-constraint* (cf. Equation (6.6)). With that, the complexity of solving a decomposed partial problem is in $O(2^{|H|})$, respectively $O(|\mathcal{T}| \cdot 2^{|H|})$ for all tasks. The definition of this partial CSOP is as follows (cf. bottom of Figure 6.3):

$$
\begin{aligned}
&\text{min.} && r_\alpha \\
&\text{s. t.} && \mathcal{C}_\tau \subseteq \mathcal{C}_\alpha, && (6.5)\\
& && \text{PAY}(\alpha) \geq \textstyle\sum_{h_\alpha \in H_\alpha^{pro}(\tau)} \text{WEIGHT}(h_\alpha) && (6.6)\\
&\text{with} && r_\alpha = |H_\alpha^{pre} \Delta\ H_\alpha^{pro}(\tau)|\,, \\
& && \mathcal{C}_\alpha \subseteq \mathcal{C} := \mathit{query\_database}(H_\alpha^{pro}(\tau)), \\
& && \mathcal{C}_\tau \subseteq \mathcal{C}, H_\alpha^{pre}, H_\alpha^{pro}(\tau) \subseteq H
\end{aligned}
$$

To aggregate all partial solutions back to a solution for the original CSOP, another optimization problem has to be solved for finding the best combination of all partial solutions. Thereby, similar to the centralized solution defined in Section 6.2.3, the number of changed hardware modules shall be minimized while allocating resources to agents. The allocation itself again is subject to a set of constraints. Similar to the centralized CSOP, the *hardware-constraint* (cf. Equation (6.8)) as well as the *task-covering-constraint* (cf. Equation (6.9)) has to hold for a valid solution. In comparison to the centralized RAP, the complexity of the aggregation

problem is heavily reduced. The *payload-constraints* of individual agents no longer have to be reviewed, as they are already taken into account in the proposal generation (cf. Equation (6.6)). Moreover, the possible resource allocations for every agent are restricted to those covered by a proposal (*proposal-constraint*, cf. Equation (6.7)). This reduces the amount of combinations from $|\mathcal{A}|^{|H|} \cdot |\mathcal{T}|^{|\mathcal{A}|}$ to $|\mathcal{T}|^{|\mathcal{A}|}$ for the aggregation problem. This problem can be defined as follows:

$$
\begin{aligned}
\text{min.} \quad & \textstyle\sum_{\alpha \in \mathcal{A}} r_\alpha \\
\text{s.t.} \quad & \forall t \in \mathcal{T}, \exists \alpha \in \mathcal{A} : H_\alpha^{pro}(t) = H_\alpha^{post}, && (6.7)\\
& \forall \alpha_1 \neq \alpha_2 \in \mathcal{A} : H_{\alpha_1}^{post} \cap H_{\alpha_2}^{post} = \emptyset, && (6.8)\\
& \exists f : \mathcal{T} \rightarrowtail \mathcal{A}, \forall t \in \mathcal{T}, \mathcal{C}_t \subseteq \mathcal{C}_{f(t)} && (6.9)\\
\text{with} \quad & r_\alpha = \left| H_\alpha^{pre} \; \Delta \; H_\alpha^{post} \right|
\end{aligned}
$$

The complexity of the decomposed approach for solving the RAP consequently is in $O(|\mathcal{T}| \cdot 2^{|H|} + |\mathcal{T}|^{|\mathcal{A}|})$, adding up the complexity for generating proposals (done in parallel for all agents) with complexity of aggregating the proposals. Due to the decomposition and distributed calculation of partial problems, it is not guaranteed that there is an optimal or even any solution for the aggregation problem. Some dependencies of the centralized CSOP are hidden for individual agents and considered the first time when aggregating the proposals, e.g., the *hardware-constraint* (resources might be allocated more often than once in different proposals). In Section 6.4 we propose an appropriate heuristic to reduce the occurrence of these dead-lock situations, among others.

6.3 Related Work

Several approaches already deal with the commonly known problem of resource allocation. Also, the idea of a distributed and market-based problem solving is widely known. However it is rather more common for solving the Task Allocation Problem [Gerkey and Matarić, 2004; Khamis et al., 2015] than the resource allocation problem [Chevaleyre et al., 2006], especially in MAS/MRS (cf. our studies on related work in Chapter 5). To the best of our knowledge, the approach we present in this thesis was the first to apply it to solve the RAP in a MAS/MRS scenario for re-allocating the capabilities of agents in such systems.

The RAP first of all is classified by the type and characteristics of resources that are subject to the allocation [Chevaleyre et al., 2006]. Following the classification of Chevaleyre et al. [2006], resources can be continuous or discrete, divisible or indivisible, sharable or non-shareable, static or dynamic, and single-unit or multi-unit. In this context, our approach for solving the RAP with limited, indivisible, not sharable, static, and multi-unit resources and executing the re-allocation online in a distributed, market-based fashion is novel, in our opinion. We now investigate other approaches for solving the RAP with market-based approaches and analyze how to how to distinguish them from our approach.

Approaches for distributed resource allocation in MAS/MRS often deal with divisible and non-sharable resources like energy [Anders et al., 2015; Wedde, 2012]. In the energy domain, the demanded amount of energy is the resource that shall be allocated to energy producers to allocate the whole resource available optimally. We know from Lai et al. [2004] that this is possible in general when dealing with dividable resources. While the optimization goal is different

from that, we examine, Anders et al. [2015] and Wedde [2012] also use market-based, distributed mechanisms to allocate resources. Because we need to deal with indivisible resources in our scenario, we cannot directly apply the approaches mentioned above. Nevertheless, their solution for solving the RAP also in a distributed fashion inspired the solution we present.

Besides applying the RAP in the energy domain, researchers also proposed approaches for applying it in the domain of distributed computing [Krauter et al., 2002]. In this field, the focus is on allocating the computing power of multiple computing nodes to certain accruing computing tasks, e.g., in high-performance computing systems [Hussain et al., 2013]. Banerjee and Hecker [2017], e.g., define the allocation of computational load needed to handle a computing task as RAP and propose a distributed approach for solving it. The goal of these approaches is to solve the RAP by minimizing the time needed to calculate a computing task and thereby is an instance of the resource scheduling problem [Kolisch and Hartmann, 1999], differentiating it from the RAP we investigate on. Because also in the domain of distributed computing, the resource to be allocated is continuous and divisible, approaches from that domain cannot be applied to our problem without necessary adaptations.

Another research field that tackles the problem of resource allocation is the one of cloud networked robotics [Kamei et al., 2012]. In the studies of Wang et al. [2017a], the resource to be allocated is the bandwidth of the network the robots work in. Like in the energy domain, the resource thus is continuous and dividable. Again, this limits the possibilities of directly applying their approach to our scenario. Other approaches define the robots themselves to be the resource of interests that other agents can allocate in the system or by humans [Wan et al., 2016]. In such application scenarios, the simultaneous allocation of more than one resource does not profoundly impact our studies. While allocating more than one resource to an agent in our studies can have positive emergent effects (the combination of two hardware modules may deliver more capabilities to an agent than the hardware modules would do alone) and negative emergent effects (the combination of two hardware modules may restrict other capabilities of the agent), [Wang et al., 2017a; Wan et al., 2016] do not take into account in their research. Because including such inter-dependencies in the calculations while generating solutions to the RAP heavily increases complexity, we cannot apply their solutions to our scenario.

Another finding of our studies in the existing literature is that when resource allocation and MAS/MRS come up together, often the Task Allocation Problem is expressed as an RAP. Rathinam et al. [2007], e.g., define the routing problem as a resource allocation problem. They define the process of reaching specific goal destinations, which should only be visited once, as tasks that need to be assigned to robots each. While doing that, robots need to keep their traveled distance (i.e., the resources they consume) minimal. This view on the system is different from ours. In our system, we need to take into account both tasks and resources at the same time and need to provide a solution for finding a task as well as a resource allocation. Thus, we cannot apply approaches like that of Rathinam et al. [2007] to our system.

The term of *self-reconfiguring robots* up to now basically was used for a field of research investigating in algorithms for optimal, efficient re-shaping (e.g., snake-bot [Thakker et al., 2014]), and self-assembly [Yim et al., 2007], most often focusing on mechanical connections of the robot modules [Østergaard et al., 2006]. Compared to the agent-controlled robots empowered with self-reconfiguration abilities of our understanding, the robots in these approaches are limited in their actual usage in real-world applications.

The way we think of self-reconfiguration, i.e., reconfiguration in terms of exchanges in the connected hardware of agents for achieving new capabilities, was investigated very sparse up to now. Preece et al. [2008] analyze such systems, where robots can be equipped with

different hardware for different tasks. In contrast to our approach, the allocation of resources is done exclusively centrally and offline, i.e., before starting the respective system. They do this by querying a static ontology that defines which capabilities are needed for specific tasks. In comparison, the solution we provide is designed to enable the system implementing it to react to changed task requirements at run-time. Moreover, Preece et al. [2008] only uses unary relationships between hardware modules and capabilities, while our approach can interpret any relationship between these two. That way, Preece et al. [2008] cannot profit from the positive emergent effects that an appropriate combination of different hardware modules can have in our approach, and further also are not able to take into account the possible negative emergent effects that obviously can arise when dealing with UAV-based systems. Thus, our approach goes beyond the considerations of Preece et al. [2008] concerning the general complexity of the RAP considering reconfigurable hardware as resources.

6.4 An Approach for a Task and Resource Allocation Strategy for Multi-Agent Systems (TRANSFORMAS)

To solve the RAP, we introduce two different algorithmic approaches. The first approach implements the procedure to solve the RAP centrally while the second does so distributively. Since distributed approaches can lead to dead-lock situations, we further present an integrated solution combining the two approaches mentioned above to mitigate this while still keeping up the possibility of profiting from the theoretical efficiency of distribution (cf. Section 6.2.4). This combined approach then acts as Task and Resource Allocation Strategy for Multi-Agent Systems (TRANSFORMAS). Additionally, we assume that for the sake of simplicity, no invalid requirements are needed in the tasks to ensure that the ensemble can solve any task it has been assigned. This can be guaranteed, for example, by a consistency check which verifies that all necessary hardware modules exist. We assume that there are no communication issues, e.g., ensured by a reliable communication infrastructure and therefore each agent can always reach all other agents.[7]

6.4.1 A Central Approach for Solving the RAP

We illustrate the process for solving the RAP with the centralized version we introduced in Section 6.2.3 within a system consisting of n agents (i.e., $|\mathcal{A}|$ agents) in the activity in Figure 6.4. The left side of the illustration shows the necessary roles adopted by the involved agents during this process of resource allocation.[8] Here, any agent can adopt the role of the coordinator (upper part) while all agents then adopt the role of participants (lower part). While all agents can adopt the role of the coordinator in general, we need to assure that for each instance of the RAP only one agent (cf. α 1 in Figure 6.4) adopts the role. Nevertheless, this agent can and should also act as a participant in the reconfiguration to include all possibilities for calculating the reconfiguration (i.e., including the hardware controlled by α

[7] like when integrating this approach into Multipotent System (cf. Section 3.2), we can achieve this by appropriate WiFi-mesh networks, proposed, e.g., by [Pojda et al., 2011; Scherer et al., 2015; Yuan et al., 2018; Stellin et al., 2020]

[8] Here, we use more general role names compared to that we use in Multipotent System. Nevertheless, the roles of *coordinator*, *participant*, *auctioneer*, and *bidder* can directly be mapped to those of RECONFIGURATION COORDINATOR, RECONFIGURATION IMPLEMENTER, RECONFIGURATION AUCTIONEER, and RECONFIGURATION PROPOSER introduced in Figure 3.3 on ENSEMBLE LAYER and AGENT LAYER

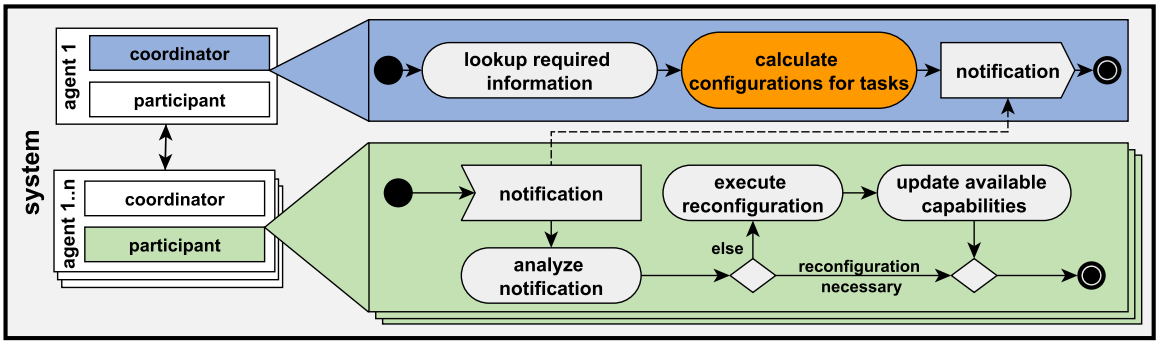


Figure 6.4: Activity for solving the resource allocation problem centrally. For a specific instance of the RAP, one agent in the system (α 1 in the figure) adopts the role of the coordinator, calculates a new resource allocation for all n participating other agents, and informs them about their new configuration. Agent α 1 and other agents (α 1, ..., n in the figure) adopt the roles of participants in that activity that realize the new system configuration afterward. Thus, also α 1 can and should participate in the process it self coordinates. The respective processes of task allocation and task execution are not represented in this figure.

1). We can achieve the distribution of roles, e.g., by an appropriate leader election algorithm [Gharehchopogh and Arjang, 2014].[9] The right side points out the different activities for the coordinator and the participants in two separated but interacting activity diagrams. When the need arises to re-allocate resources in the system as a task allocation failed in advance (cf. Figure 6.2), the system autonomously initiates the adaption process. After the coordinator has been elected, it collects the necessary data for solving the RAP. Since we intend the approach we present in this section to be a centralized one, we also can assume that the current global system configuration, as well as the current requirements to the RAP every agent knows. Thus, the first component of this information is on individual agents, e.g., their current configuration concerning hardware and their maximum load capacity. The second component of this information describing the concrete instance of the RAP then consists of the task requirements in terms of the required capabilities of every one of these tasks. After the required data collection is completed, the coordinator centrally solves the RAP as defined in Section 6.2.3. The overall solution now contains the new configurations comprised of hardware modules allocated to agents, which the coordinator distributes to all available participants. As soon as the participant receives the new resource allocation message, it analyzes it by comparing its current hardware configuration with the new configuration. Additionally, it saves the new (current) configuration of all other agents to keep the global knowledge consistent. Then it decides if a reconfiguration is necessary or not. If so, an actual reconfiguration is executed, leading to a change of hardware modules. Each agent is now equipped with new capabilities. If an agent already possesses the required hardware modules, the reconfiguration is complete without any hardware modifications for this participant. The same process is performed on all participants involved in parallel. In a final step, all participants notify the coordinator that they have finished their reconfiguration, and the overall system reconfiguration sequence is finished by

[9]While we do not further focus on the topic of leader election in this chapter, we can integrate it in the process we describe for Multipotent System in Section 3.2.

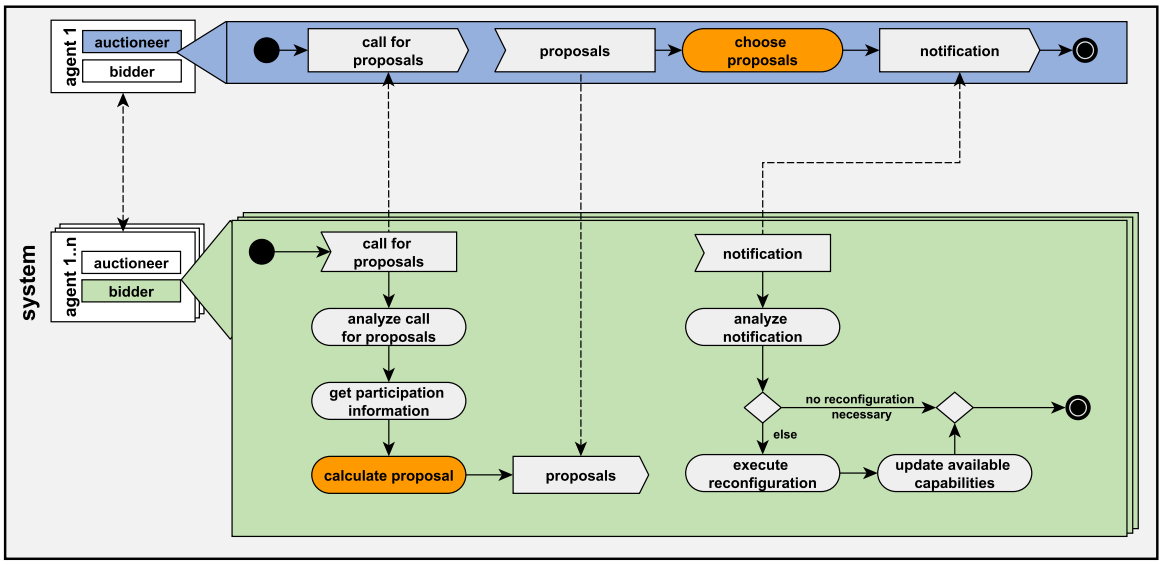


Figure 6.5: Activity for solving the resource allocation problem distributively. One agent acts as auctioneer initiating a market-based resource allocation with a Call for Proposals (CfP). When receiving proposals from participating agents (bidders) it aggregates them to a new system configuration solving the RAP, if possible. Each bidder calculates only partial solutions to the RAP (cf. Figure 6.3). Similar to the centralized solution, bidders realize new system configurations when they get informed by the auctioneer. Again, nether task allocation nor execution are represented.

the coordinator (not included in Figure 6.4).

6.4.2 A Distributed Approach for Solving the RAP

Depending on the application scenario, the number of hardware modules, tasks, and agents can be very large. To counteract this complexity, we propose to decompose the RAP by calculating partial solutions and aggregating them to find the solution that most closely resembles the central solution as formalized in Section 6.2.4. Analogously to Figure 6.4 for the central approach, Figure 6.5 is separated into the same two parts for the distributed approach. As the problem is solved using a market-based approach, we introduce additional roles for the auction. In addition to its usual role, the coordinator adopts the auctioneer's role while the participants also become bidders. Just like in the central case, every agent can take on any role in principle, so first of all, a leader election is performed to determine the roles. As mentioned in Section 6.2.4, the objects up for auction are the tasks for which proposals containing the hardware modules that the bidder would like to allocate for each task are generated. The auctioneer sends a call for proposals to start the auction, which consists of the required tasks, including their required capabilities. After receiving the call for proposals, the bidder analyzes it and collects the pertinent information. This entails information on all existing available hardware in the system (but not its current allocation), the individual agent's maximum payload, and other local dependencies, e.g., between hardware types and capabilities. As soon as all required information has been collected, the bidder calculates a

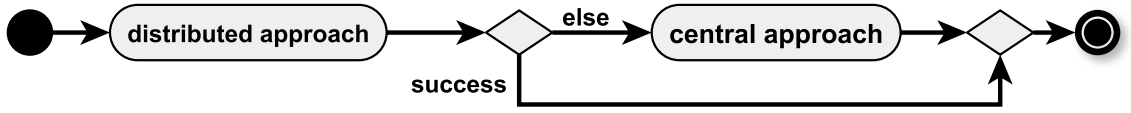


Figure 6.6: Activity for our TRANSFORMAS approach. If no solution for the RAP can be determined distributively, a solution is calculated centrally.

bid based on its local knowledge by solving the CSOP for proposals in Section 6.2.4 for every task of the RAP and sums it up as a single proposal that is sent to the auctioneer. Each bid contains the hardware modules that the bidder would like to employ for the associated task, i.e., a partial resource allocation as a partial solution of the RAP. These steps are carried out similarly by all other bidders. Once the auctioneer has received all proposals, it aggregates all these partial solutions of the RAP back to a complete solution by solving the aggregating CSOP of Section 6.2.4. All bidders are notified by sending a message with the new configuration, which only contains the accepted bid, including the participants' new hardware configuration instead of the whole new system configuration as needed in the central approach. From this step on, the activity follows the same procedure as in the centralized approach.

6.4.3 The Integrated Solution TRANSFORMAS

As described in Section 6.2.4, each agent only considers its configuration and available hardware modules when solving the partial RAP distributedly, instead of considering all changes by other agents when minimizing the number of hardware reconfigurations. Due to this decomposition, some dependencies of the centralized CSOP are not apparent to the individual agent, e.g., specific hardware modules might be allocated more often than once in different proposals (*hardware-constraint*). As soon as the partial solutions are aggregated, it is possible that no solution can be found, and a deadlock situation occurs. We circumvent this problem by requesting more than one proposal from every bidder, each of them unique concerning the requested hardware for a specific task to increase the number of possible allocations. To enable the submission of multiple bids for the same task, bids of declining cost-effectiveness concerning r_α (k-best) must be calculated. Thus, the first proposal contains the best local solution, in which as little as possible, plugging and unplugging processes occur. The second proposal represents the second-best, locally calculated solution and so on. To allow all bidders to submit more than one proposal, the auctioneer additionally sends the desired number of proposals (k). Depending on k, the bidder calculates up to k proposals, which are then sent to the auctioneer in the next step. The sequence follows the same procedure as described in the distributed approach. As this is still a heuristic approach, which does not guarantee a solution even if one exists, we overcome possible dead-lock situations by integrating the distributed approach with the centralized approach (cf. the activity in Figure 6.6), we call Task and Resource Allocation Strategy for Multi-Agent Systems (TRANSFORMAS). After the distributed approach fails to solve the RAP, additional communication takes place to provide the auctioneer with local information about all bidders to solve the RAP centrally. After the communication is completed the RAP is solved with the central approach analogously to Section 6.4.1.

6.5 Evaluation

In our evaluation, we want to examine our approaches from Section 6.4 and compare them under different conditions. Therefore, we are interested in analyzing relevant properties of the central approach (*CA*, cf. Section 6.4.1) and the distributed approach (*DA,* cf. Section 6.4.2) concerning *1) the total time* consumed for computing a new resource allocation, *2) the quality* of a solution concerning needed hardware changes (cf. $\sum_{a \in A} r_a$ in Section 6.2.3), and *3) the success rate*, i.e., how often is a solution to the RAP found. TRANSFORMAS introduced in Section 6.4.3 is not subject to the comparison as it combines the aforementioned CA and DA in a promising way. Instead, we investigate our k-best heuristic for improving the success rate and quality. We state the following hypotheses we want to verify in the following:

$\mathcal{HYP}_1$) DA outperforms CA in terms of computation time needed, especially with increasing problem size.

$\mathcal{HYP}_2$) The quality of solutions calculated by DA is equal to that calculated by CA.

$\mathcal{HYP}_3$) The success rate of DA is higher than that of the CA within a defined time limit.

$\mathcal{HYP}_4$) Increasing k in our k-best heuristic increases the success rate and solution quality of TRANSFORMAS.

6.5.1 Testbed

For both CA and DA, we define a maximum timeout for calculating valid solutions of 300 seconds. We can justify that decision because we intend to deploy our approach to real hardware, e.g., flying ensembles, where immense timeouts are undesirable. If calculating a solution for the RAP needs more than those 300 seconds, we typically generate more problems for the system' user caused by reduced power for operating the respective hardware (e.g., an UAV) than we solve by autonomously calculating a new and beneficial configuration concerning their connected hardware. This timeout is of high relevance for measuring the success rate as we only accept those solutions as valid ones that were calculated before it. The success rate also is influenced by the way we create our evaluation scenarios.

We evaluated the properties of our algorithms for solving the RAP in a modularized component enabling us to analyze them isolated from other mechanisms, e.g., task allocation, leader election, et cetera. Instead, we decided to determine initial conditions (i.e., agent and hardware properties and task requirements) randomly but equally for the comparative evaluations. This, of course, does not guarantee that, in general, there is a valid solution for every one of these initial configurations (cf. Section 6.4) which is undesirable for our comparative evaluations. Therefore, we cannot evaluate the results from our approaches to a "ground-truth" but only against each other. That means that for a given initial condition, we say a RAP is solvable if and only if one of our approaches (including all heuristics) can find a solution within the allowed time frame. The ground-truth cannot be determined otherwise as this would need the central solver to either prove a CSOP to be solvable or insolvable, which often cannot be computed in adequate time as the problem itself is NP-hard [Chevaleyre et al., 2006]. While according to this, the number of total runs performed per configuration differs (cf. Tables 6.1 to 6.3), we ensured to have at least 100 valid runs per problem size (except for one) to achieve significant results.

The problem size of the RAP we study is determined by the number of participating agents $|\mathcal{A}|$, the number of tasks $|\mathcal{T}|$ defining the resource requirements, the number of capabilities

$|\mathcal{C}|$ a task can require, and the number of hardware modules $|H|$, randomly instantiated of different types (i.e., providing different capabilities). For identifying the individual effect on the properties we investigate in hypothesis $\mathcal{HYP}_1$ – $\mathcal{HYP}_4$, we systematically increase each of these parameters and create an evaluation run for every reasonable combination of parameters. To reduce the complexity in our evaluations, we restrict our MRTA to SR-ST as we describe it in Sections 6.2 and 6.3. Consequently, we exclude combinations containing parameters set to useless values, e.g., combinations containing $|\mathcal{T}| > |\mathcal{A}|$ or $|\mathcal{T}| > |H|$ are excluded. In our experimental runs, we iteratively increase the problem size in terms of $|\mathcal{A}|$, $|\mathcal{T}|$, and $|\mathcal{C}|$ in steps of 2 in the integers interval $[2, 10]$ (the limit of 10 assures feasibility also for mobile hardware while still being adequate for our problem domain) while keeping $|H| = 10$ (to assure there are enough modules for the requirements). We perform *paired t-tests* (one- or two-sided, where indicated) for normally distributed populations to support our hypothesis. For non-normal distributed populations, we used *Mann-Whitney-U tests*, respectively. To verify the type of distribution in the population, we perform *Kolmogorov-Smirnov tests* for distinguishing non-normal distributed populations from normally distributed ones. We performed the evaluations we describe in Section 6.5.2 with high-performance server hardware (16 core @3GHz CPU, 32 GB RAM). Evaluations, we describe in Section 6.5.3, we instead performed with computing hardware that we can also use for controlling flying ensembles (an Odroid XU4 [Hardkernel, 2018] with an Exynos5422 octa-core ARM processor and 2 GB of RAM)[10] To find solutions for the RAP (cf. Section 6.2), we use Gecode as a solver for the CSOPs which we modeled with MiniZinc.[11] We do not take into account messaging overhead in our run time measurements as they require a static amount of time(CA needs $|\mathcal{A}|$ messages, DA needs $3 \cdot |\mathcal{A}|$).

6.5.2 Results

To support our hypotheses $\mathcal{HYP}_1$ – $\mathcal{HYP}_3$ we compare results achieved from CA with that of the DA with $k = 1$ (cf. Tables 6.1 to 6.3, columns labeled with $k = 1$), while for supporting $\mathcal{HYP}_4$ we investigate in columns with $k = 2$ and $k = 3$ also. In Tables 6.1 to 6.3, problem sizes are encoded as, e.g., *10:8:6:4* for $|\mathcal{A}| = 10$, $|H| = 8$, $|\mathcal{T}| = 6$, $|\mathcal{C}| = 4$. We further classify the problem sizes by selecting one of these parameters $(\mathcal{A}, H, \mathcal{T}, \mathcal{C})$ to be either set to a *great* or *low* number and whether the problem size defined by the other parameters is *small* or *big*[12], e.g., analyzing the parameter agents $(\mathcal{A})$ set to a low $(\mathbf{L} = 4)$ size while other parameters define a big $(\mathbf{B} = |\mathcal{A}|{:}10{:}4{:}4)$ problem is abbreviated as **ABL**.

6.5.2.1 Investigating $\mathcal{HYP}_1$: DA Outperforms CA in Terms of Computation Time Needed, Especially with Increasing Problem Size.

To support our hypothesis we compare run times for different problem sizes picked from Table 6.1 where we increase the parameter under observation from *low* to *great* and the problem size from *small* to *big* respectively. To achieve a fair comparison, we neglect results originating from initial conditions where DA determines no valid solution. As those situations are uncovered very quickly, results would be falsified in terms of reducing the mean run time of

[10] the single-board computer were the same we used for controlling the prototypes in our field exeperiments for demonstrating the feasibility of implementing and deploying Multipotent System for real world systems (cf. Sections 3.4.1 and 3.4.2.

[11] MiniZinc on `http://www.minizinc.org/`, and Gecode on `http://www.gecode.org/`

[12] *Small / big*: We set other parameters to the lowest / highest possible values.

Table 6.1: Comparative evaluation of the central approach (CA) and distributed approach (DA) and evaluation of the k-best-algorithm of TRANSFORMAS. We compare run-time needed by CA and DA (for each $k \in \{1, 2, 3\}$), measuring those finishing within our time limit. For each comparison CA and DA work on equal problems. This is also ensured for the k-best evaluation (equal problems for each k compared).

problem size $\mathcal{A}$:H:$\mathcal{T}$:$\mathcal{C}$	**# total runs**	**run-time in milliseconds: mean (std)**					
		k=1		*k=2*		*k=3*	
		CA	DA	CA	DA	CA	DA
ASL *2:10:2:2*	183	172 (8)	426 (26)	173 (8)	952 (58)	173 (8)	1,272 (246)
ASG *10:10:2:2*	199	1,419 (2,958)	612 (114)	1,710 (3,573)	2,873 (4,544)	1,751 (3,620)	3,952 (5,136)
ABL *4:10:4:4*	1,138	2,202 (8,707)	893 (140)	2,219 (6,727)	2,143 (240)	2,219 (6,727)	3,247 (532)
ABG *10:10:4:4*	274	258,744 (99,323)	1,136 (200)	248,413 (107,739)	70,768 (92,141)	240,998 (114,806)	70,065 (80,131)
ASL *2:10:2:2*	183	172 (8)	426 (26)	173 (8)	952 (58)	173 (8)	1,272 (246)
TSG *6:10:6:2*	618	123,731 (109,208)	1,167 (165)	141,809 (120,748)	20,962 (10,279)	141,809 (120,748)	44,227 (16,080)
TBL *10:10:2:4*	196	12,939 (41,173)	699 (118)	17,199 (51,855)	5,710 (10,388)	18,812 (55,935)	6,206 (13,932)
TBG *10:10:6:4*	71	300,375 (43)	1,715 (291)	300,375 (29)	75,024 (88,875)	300,381 (28)	118,401 (100,551)
HSL *2:2:2:2*	15,465	125 (7)	384 (52)	125 (6)	809 (15)	125 (6)	1,015 (197)
ASL *2:10:2:2*	183	172 (8)	426 (26)	173 (8)	952 (58)	173 (8)	1,272 (246)
HBL *10:4:4:4*	378	45,130 (67,567)	976 (167)	45,245 (70,612)	13,233 (18,664)	45,245 (70,612)	20,657 (27,895)
ABG *10:10:4:4*	274	258,744 (99,323)	1136 (200)	248,413 (107,739)	70,768 (92,141)	240,998 (114,806)	70,065 (80,131)
ASL *2:10:2:2*	183	172 (8)	426 (26)	173 (8)	952 (58)	173 (8)	1,272 (246)
CSG *2:10:2:10*	1,124	18,478 (64,623)	590 (165)	18,381 (64,527)	1,284 (328)	18,381 (64,527)	1,512 (555)
CBL *10:10:4:2*	316	210,576 (130,356)	967 (185)	158,513 (145,970)	110,510 (108,538)	87,800 (133,046)	71,972 (77,058)
CBG *10:10:4:10*	281	291,803 (44,311)	1,613 (408)	290,400 (46,168)	38,199 (63,476)	291,129 (45,389)	42,791 (69,047)

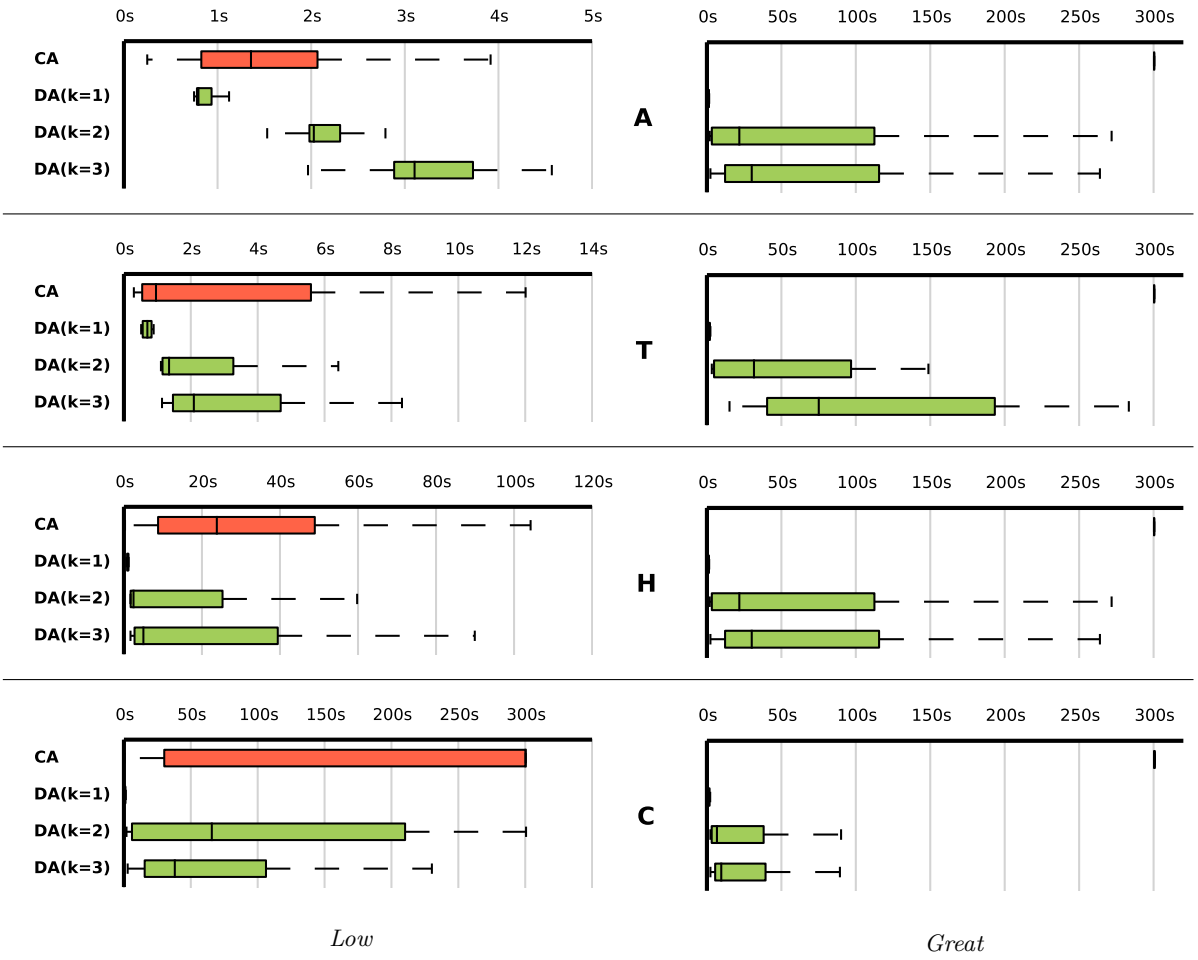


Figure 6.7: Run time comparison of CA and DA (k=1,2,3) in *big* problems (box plots for time in seconds). We scale x-axis dynamically for each problem size.

DA while increasing that of the CA. This also explains varying average run times for CA in Table 6.1 for $k = 1, 2, 3$. In *small* configurations CA outperforms DA in every configuration significantly when the parameter of relevance is set to low (e.g., ASL: CA needs 172ms, DA needs 426ms, $p = 7.1 \cdot 10^{-215}$, or HSL: CA needs 125ms, DA needs 384ms, $p = 0.0$). This can be explained due to the natural overhead (proposal generation and aggregation), DA suffers from, and CA does not. When increasing to the respective *great* problem, this relation is reversed completely. While the run time for CA increases from ASL to ASG with factor 8.25, DA increases only with factor 1.47. The same relation holds for CSL to CSG (for CA with factor 107.43, for DA with factor 1.38) and also for TSL to TSG (CA with factor 720.34, DA with factor 2.74). Thus, in these *great* configurations, DA outperforms CA significantly. Only when increasing $|H|$ from HSL to HSG has less influence on CA's run-time (factor 1.38, factor for DA is 1.11, cf. ASL). CA's performance is even worse in big problems and already close to the calculation limit within our time out (we consider timed out runs with 300s in the average calculation, which is an underestimation). While *low* numbers for $|\mathcal{A}|$ (ABL needs 2.20s), $|\mathcal{T}|$ (TBL needs 12.94s) and $|H|$ (HBL needs 45.13s) are still feasible within our time limit (cf. CA in Figure 6.7 left), increasing the parameter of interest to *great* (cf. CA in Figure 6.7 right)

leads to impractical run times of 258.74s in ABG (factor 117.5), 300.385s in TBG (factor 23.2), and 258.74s in HBG (factor 5.73). For $|\mathcal{C}|$ increasing from CBL to CBG seems to not have that huge influence on run time (factor 1.38) but this is only due to the time out restriction and the already high run time for CBL (212.58s) and thus run times for CBL and CBG are impractical (cf. Figure 6.7). Like in *small* problems DA again is very robust against increasing from *low* to *great* in *big* problems for almost all parameters. The biggest increase in run time is caused by $|\mathcal{T}|$ (from TBL with 699ms to TBG with 1715ms, factor 2.45), but for all problem sizes the distributed approach's run time is below 2000ms thus significantly lower than CA for all *big* problems in Table 6.1. We see that especially with increasing the problem size, the DA outperforms CA by orders of magnitude significantly (*Cohens's d test* results in values from approx. $258,000 - 300,000$ in *big* problems in Table 6.1) and thus $\mathcal{HYP}_\mathbf{1}$ **holds**. This is also true for all problems bigger than $|\mathcal{A}| \geq 6, |H| \geq 6, |\mathcal{T}| \geq 4, |\mathcal{C}| \geq 4$ where we see a mean run time (we do not depict that aggregated value in Table 6.1) for CA vs. DA of $1,016ms$ ($3,192ms$) to $575ms$ ($166ms$) which is significant ($p = 2.30 \cdot 10^{-5}$).

6.5.2.2 Investigating $\mathcal{HYP}_\mathbf{2}$: The Quality of Solutions Calculated by DA is Equal to that Calculated by CA.

To support $\mathcal{HYP}_\mathbf{2}$, we compare the results from different problem sizes. Again, we want to perform fair comparisons and thus only consider results originating from initial conditions where both approaches achieved a valid solution. By analyzing our results in Table 6.2, we cannot find any significant difference between CA and DA (all data sets are normally distributed, so we use *t-tests*). Differences occurring in ABL ($p = 0.35$), HBL ($p = 0.84$), and CSG ($p = 0.94$) are only by chance. Thus, we find that if DA achieves a solution, its quality is equal to that CA would have found, and $\mathcal{HYP}_2$ **holds**. For generalizing this statement, we also evaluated further problem sizes but never identified a significant difference. In this comparison, we see the lowest confidence for *4:8:2:2* with CA = 0.73, DA = 0.86, and $p = 0.11$, where we could not rejected the hypothesis as we test for equality here.

6.5.2.3 Investigating $\mathcal{HYP}_\mathbf{3}$: The Success Rate of DA is Higher than that of the CA Within a Defined Time Limit.

To measure the success rate, we only consider those runs performed on initial conditions where we know a valid solution exists (cf. Section 6.5.1). In Table 6.3 within columns *success* the rate is given as a ratio calculated by the total amount of solutions found by an approach divided by the number of total runs in the first column. In the second column, the improvement to the approach to the left is given in absolute numbers (+ indicates that, e.g., DA found solutions where CA did not and − vice versa) and in relative numbers. For *small* problem sizes, we see that CA is capable of finding valid solutions in almost every run, indicated by a success rate of 1 (we only see minor dropouts with 0.91 in TSG and 0.97 in CSG and CBL). Compared to this, DA achieves a much more diverse spread success rate, ranging from 0.03 in TSG (where the central approach achieves 0.91) to 0.92 in TBL (central achieves 0.97 here). With a look at improvement from CA to DA in *small* problems, on the one hand, the biggest improvement can be seen in CSG, where DA solves 33 problems CA could not, while there are 213 problems that on the other hand cannot be solved. For all *small* problems, the improvement concerning DA over CA is below 1.0. This indicates, that for *small* problem sizes $\mathcal{HYP}_3$ must be rejected. In *big* problem sizes, this is no longer the case, as the CA cannot hold

Table 6.2: Comparative evaluation of the central approach (CA) and distributed approach (DA) and evaluation of the k-best-algorithm of TRANSFORMAS. We compare the quality of solutions concerning the amount of needed plug-in and plug-off processes. For each comparison CA and DA work on equal problems. This is also ensured for the k-best evaluation (equal problems for each k compared).

problem size $\mathcal{A}$:H:$\mathcal{T}$:$\mathcal{C}$	**# total runs**	**quality in needed reconfigurations: mean (std)**								
		k=1			*k=2*			*k=3*		
		CA	k=1	delta	k=1	k=2	delta	k=2	k=3	delta
ASL	183	0.77	0.77	0.00	0.77	0.77	0.00	0.77	0.77	0.00
2:10:2:2		(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)
ASG	199	0.68	0.68	0.00	0.64	0.64	0.00	0.65	0.65	-0.04
10:10:2:2		(0.65)	(0.65)	(0.0)	(0.64)	(0.64)	(0.0)	(0.64)	(0.64)	(0.31)
ABL	1,138	2.73	2.80	-0.07	2.73	2.75	0.04	2.73	2.75	0.00
4:10:4:4		(0.99)	(1.07)	(0.35)	(0.99)	(1.03)	(0.3)	(0.99)	(1.03)	(0.0)
ABG	274	1.32	1.32	0.00	0.28	2.24	0.12	0.31	2.21	-0.12
10:10:4:4		(0.69)	(0.69)	(0.0)	(0.62)	(0.85)	(0.53)	(0.65)	(0.86)	(0.62)
ASL	183	0.77	0.77	0.00	0.77	0.77	0.00	0.77	0.77	0.00
2:10:2:2		(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)
TSG	618	3.94	3.94	0.00	3.19	3.86	0.00	3.19	3.86	0.00
6:10:6:2		(0.8)	(0.8)	(0.0)	(1.71)	(0.89)	(0.0)	(1.71)	(0.89)	(0.0)
TBL	196	1.03	1.03	0.00	1.01	1.02	0.00	1.01	1.02	-0.03
10:10:2:4		(0.73)	(0.73)	(0.0)	(0.74)	(0.73)	(0.0)	(0.73)	(0.73)	(0.23)
TBG	71	-	-	-	0.00	7.00	0.00	-	-	0.00
10:10:6:4		-	-	-	(0.0)	(0.0)	(0.0)	-	-	(0.0)
HSL	15,465	1.26	1.26	0.00	1.26	1.26	0.00	1.26	1.26	0.00
2:2:2:2		(0.67)	(0.67)	(0.0)	(0.67)	(0.67)	(0.0)	(0.67)	(0.67)	(0.0)
ASL	183	0.77	0.77	0.00	0.77	0.77	0.00	0.77	0.77	0.00
2:10:2:2		(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)
HBL	378	2.79	2.81	-0.02	2.64	2.82	0.00	2.64	2.82	0.00
10:4:4:4		(0.59)	(0.65)	(0.18)	(0.85)	(0.64)	(0.0)	(0.85)	(0.64)	(0.0)
ABG	274	1.32	1.32	0.00	0.28	2.24	0.12	0.31	2.21	-0.12
10:10:4:4		(0.69)	(0.69)	(0.0)	(0.62)	(0.85)	(0.53)	(0.65)	(0.86)	(0.62)
ASL	183	0.77	0.77	0.00	0.77	0.77	0.00	0.77	0.77	0.00
2:10:2:2		(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)	(0.71)	(0.71)	(0.0)
CSG	1,124	1.99	1.99	0.00	1.92	2.00	0.00	1.92	2.00	0.00
2:10:2:10		(1.26)	(1.27)	(0.08)	(1.3)	(1.26)	(0.0)	(1.3)	(1.26)	(0.0)
CBL	316	1.40	1.40	0.00	0.66	1.48	0.06	0.82	1.24	0.00
10:10:4:2		(0.81)	(0.81)	(0.0)	(0.56)	(0.68)	(0.38)	(0.45)	(0.63)	(0.0)
CBG	281	1.80	1.80	0.00	0.08	3.00	0.04	0.07	3.02	-0.09
10:10:4:10		(0.98)	(0.98)	(0.0)	(0.42)	(0.92)	(0.37)	(0.4)	(0.93)	(0.57)

Table 6.3: Comparative evaluation of the central approach (CA) and distributed approach (DA) and evaluation of the k-best-algorithm of TRANSFORMAS. We compare the success rate concerning the amount of valid solutions found within our time limit. For each comparison CA and DA work on equal problems. This is also ensured for the k-best evaluation (equal problems for each k compared).

problem size $\mathcal{A}$:H:$\mathcal{T}$:$\mathcal{C}$	**# total runs**	**success: absolute (rate)** and **improvement absolute (relative)**						
		CA	*k=1*		*k=2*		*k=3*	
		# (rate)	# (rate)	± (rel)	# (rate)	± (rel)	# (rate)	± (rel)
ASL *2:10:2:2*	183	183 (1.0)	129 (0.7)	0/54 (0.7)	158 (0.86)	29/0 (1.22)	158 (0.86)	0/0 (1.0)
ASG *10:10:2:2*	199	199 (1.0)	187 (0.94)	0/12 (0.94)	183 (0.92)	11/15 (0.98)	186 (0.93)	3/0 (1.02)
ABL *4:10:4:4*	1,138	1,138 (1.0)	391 (0.34)	0/747 (0.34)	733 (0.64)	342/0 (1.87)	733 (0.64)	0/0 (1.0)
ABG *10:10:4:4*	274	57 (0.21)	199 (0.73)	168/26 (3.49)	159 (0.58)	57/97 (0.8)	115 (0.42)	4/48 (0.72)
ASL *2:10:2:2*	183	183 (1.0)	129 (0.7)	0/54 (0.7)	158 (0.86)	29/0 (1.22)	158 (0.86)	0/0 (1.0)
TSG *6:10:6:2*	618	563 (0.91)	21 (0.03)	4/546 (0.04)	181 (0.29)	160/0 (8.62)	181 (0.29)	0/0 (1.0)
TBL *10:10:2:4*	196	190 (0.97)	180 (0.92)	3/13 (0.95)	174 (0.89)	13/19 (0.97)	175 (0.89)	4/3 (1.01)
TBG *10:10:6:4*	71	4 (0.06)	42 (0.59)	42/4 (10.5)	26 (0.37)	25/41 (0.62)	6 (0.08)	0/20 (0.23)
HSL *2:2:2:2*	15,465	15,465 (1.0)	9,617 (0.62)	0/5,848 (0.62)	12,648 (0.82)	3,031/0 (1.32)	12,648 (0.82)	0/0 (1.0)
ASL *2:10:2:2*	183	183 (1.0)	129 (0.7)	0/54 (0.7)	158 (0.86)	29/0 (1.22)	158 (0.86)	0/0 (1.0)
HBL *10:4:4:4*	378	361 (0.96)	128 (0.34)	7/240 (0.35)	294 (0.78)	166/0 (2.3)	294 (0.78)	0/0 (1.0)
ABG *10:10:4:4*	274	57 (0.21)	199 (0.73)	168/26 (3.49)	159 (0.58)	57/97 (0.8)	115 (0.42)	4/48 (0.72)
ASL *2:10:2:2*	183	183 (1.0)	129 (0.7)	0/54 (0.7)	158 (0.86)	29/0 (1.22)	158 (0.86)	0/0 (1.0)
CSG *2:10:2:10*	1124	1,089 (0.97)	909 (0.81)	33/213 (0.83)	976 (0.87)	67/0 (1.07)	976 (0.87)	0/0 (1.0)
CBL *10:10:4:2*	316	125 (0.4)	230 (0.73)	153/48 (1.84)	123 (0.39)	56/163 (0.53)	57 (0.18)	0/66 (0.46)
CBG *10:10:4:10*	281	15 (0.05)	234 (0.83)	224/5 (15.6)	229 (0.81)	46/51 (0.98)	213 (0.76)	4/20 (0.93)

its success rate. It especially suffers from increasing $|\mathcal{A}|$, respectively $|H|$. While the success rate is still 1.0 for ABL and 0.96 in HBL, it drops to 0.21 in ABG/ HBG). The same effect can be seen concerning $|\mathcal{C}|$. From 0.4 for CBL, the success rate drops to 0.05 in CBG. In contrast, DA is not influenced that hard by switching from *low* to *great* in *big* problems. The biggest drop can be seen from TBL with 0.92 to TBG with 0.59, which is a challenging problem when considering that CA only successes with a rate of 0.06. For ABL, the success rate of 0.34 (and for HBL 0.34 respectively) does not drop, but increases instead in ABG/ HBG to 0.73 which compared to the CA is an improvement of $+168/-26$ in absolute numbers (i.e., 349%). The same can be seen for CBL with an improvement from 0.73 to CBG with 0.83 where especially the improvement from CA to DA of $+224/-5$ (i.e., 1560%) is remarkable. This effect can be lead back to the increased flexibility through an increased amount of proposals in ABL/ ABG and to the reduced chance for resource conflicts in HBL/ HBG. To support our hypothesis, we made some further evaluations, including all other combinations in *big* problems, to get a general statement where we compare the success rate of the CA to DA. The results is that the overall average success rate for CA is 0.04, for DA 0.6, which is a significant difference ($p = 2.5 \cdot 10^{-6}$) and thus $\mathcal{HYP}_3$ **holds** for *big* problems in general (like in Table 6.3 we define big problems as $|\mathcal{A}| = 10, |H| = 10, |\mathcal{T}| \geq 4, |\mathcal{C}| \geq 4$). There is still a significant difference for $|\mathcal{A}| \geq 8, |H| \geq 6, |\mathcal{T}| \geq 4, |\mathcal{C}| \geq 4$ with CA (0.24) vs. DA (0.48) with ($p = 5.2 \cdot 10^{-6}$). Reducing $|\mathcal{A}|$ to 6 seems to be the cutting point (CA 0.43 vs. DA 0.42) with ($p = 0.84$) where we can not see a significant difference any more.

6.5.2.4 Investigating $\mathcal{HYP}_4$: Increasing k in Our k-Best Heuristic Increases the Success Rate and Solution Quality of Transformas.

We see that the success rate is very diversely influenced by increasing k in our k-best algorithm. On the one hand, Tables 6.1 to 6.3 show that increasing k from 1 to 2 has a positive effect especially in *small* problems. For ASL, e.g., we see an improvement from 0.7 to 0.86 with 29 additional (0 less) problems solved or even more in TSG (from 0.03 to 0.29) or HSL (0.62 to 0.82). But also in *big* problems $k = 2$ improves success rate in certain configurations, e.g., in ABL from 0.34 to 0.64 and in HBL from 0.34 to 0.78. On the other hand, we also see heavily decreasing success rates, e.g., in ABG (from 0.73 to 0.58) or CBL (from 0.73 to 0.39). This drop effect is due to the increased combinatorial problem of aggregating partial problems, which often is not possible within the timeout (we see, e.g., in CBL the mean run time of $k = 2$ compared to $k = 1$ increases from less than $1s$ to $110s$ and a standard deviation of $108s$). Increasing $k = 2$ to $k = 3$ does not seem to bring any benefit, again due to heavily increased run times preventing a solution from being determined before the time out. Some problems are even worse handled by $k = 3$ compared to $k = 2$, e.g., for CBL, we see a drop in success rate from 0.39 to 0.18 with a total of 66 less solved problems. For all problem sizes included in Table 6.2 we cannot find any significant improvement from $k = 1$ to $k = 2$ nor from $k = 2$ to $k = 3$ concerning solution quality, but in contrast see an increased amount of needed reconfigurations (this still holds for all other evaluated problem sizes). Thus $\mathcal{HYP}_4$ only **partially holds** concerning success rate and has to be rejected for quality improvements. Consequently, we intend that for TRANSFORMAS, a portfolio approach for selecting the appropriate size of k for specific problem sizes works the best.

6.5.3 Further Results

To validate real-world applicability, we further evaluate *1) the scalability of DA* in problem sizes similar to that described in Section 6.1 and *2) the deployment on hardware we can use for flying ensembles involving UAV*.

For *1)* we investigate in problems of size *10:h:10:6* with $h \in \{48, 96, 192\}$. The run time of DA in such problems still stays in a tolerable frame of $12.16s$ ($9.72s$) for $h = 48$, $34.37s$ ($27.41s$) for $h = 96$, and $87.89s$ ($64.26s$) for $h = 192$. Remarkably, the success rate of DA in those problems is always 1.0. This can be explained by the correlation between the increased amount of available hardware and the variance in allocated hardware within proposals. This induces that for large-scale problems (concerning $|H|$) DA is very well suited. Evaluation of other problems increasing $|\mathcal{A}|$ show that this is not longer true for $|\mathcal{A}| > 10$. Those sizes need further decomposition strategies we want to investigate in our future research. For *2)* we deploy TRANSFORMAS on our robot hardware and evaluate ABG, TBG, HBG, and CBG from Section 6.5.2. We find that those problems are also feasible within appropriate time (ABG/ HBG: $3.47s$ ($0.60s$), TBG: $4.88s$ ($0.93s$), CBG: $4.93s$ ($0.70s$)) with success rates between 0.83 and 0.92. Compared to the results in Section 6.5.2 we see a hardware-related increase ranging from 285% to 306%. Thus, our approach also passes the reality check.

6.6 Future Research Directions

In future experiments, we can further extend the models used for individual robots to respect increased heterogeneity in the hardware we consider, e.g., individual hardware-capability relationships resulting from heterogeneous geometric designs of hardware modules. We assume that we can integrate such dependencies into our current model easily. Because such extensions further increase the complexity of the CSOP model, we expect they will further emphasize the advantages of our distributed approach we already detected in Section 6.5 when increasing the problem size.

Further, we already achieved some first results concerning the real-world implementation of our robots equipped with the ability to autonomously reconfigure their capabilities that we plan to investigate further.

Moreover we can think of dropping the goal of optimality and instead focus on efficiency, e.g., by applying a greedy algorithm for solving the RAP. Such solution could be beneficial in large systems involving even more agents and hardware modules or when the plan size increases heavily in terms of tasks or respectively required capabilities.

Chapter Summary and Outlook

In this chapter, we analyzed situations were a MAS/MRS cannot handle a user-defined mission because of an insufficient configuration of its agents concerning their hardware composition. We define this problem as an instance of the Resource Allocation Problem (RAP) and provide multiple solutions for solving it centrally and distributedly by mapping it to a Constraint Satisfaction (and Optimization) Problem (CSOP) each. Within the CSOP we introduce for solving the RAP, we can easily respect the internal properties of the agents involved, which are relevant for re-allocating resources, e.g., an agent's maximum payload or individual geometric designs. We can find such internal inter-dependencies frequently when dealing with flying ensembles like those we aim at with our reference architecture for Multipotent Systems. For solving the resource allocation problem, we propose Transformas. Transformas is an algorithmic approach that decomposes and solves the RAP in a distributed fashion if possible. In our evaluations of these solutions, we found that the distributed solution outperforms the central solution in most instances of the problem. This finding holds significantly when we increase the problem size in terms of the agents in the system, the tasks we require in a plan, the number of available hardware in the system, or the number of capabilities we require in the individual tasks. Furthermore, and to the best of our knowledge, we provide the first approach to adapt the capabilities provided by robots at run-time. We thereby overcome the need for specialization, typically restricting the applicability of MAS/MRS in other current approaches. While we perform the analysis having Multipotent System as one representative in mind, we designed and modeled our solution to be applicable by other systems facing the same situation. In Multipotent System, nevertheless, we can use this solution when the configuration of agents in the system does not comply with the tasks included in a plan resulting from a SCORe mission we introduced as the Multipotent System's user. When we deploy Multipotent System to flying ensembles, we face such situations often because of the physical limitation a flying agent has to deal with concerning its possible additional payload. Then, one agent typically cannot be equipped with all hardware necessary for providing all capabilities required in each of the possible tasks. Thus, when we did not configure the system correctly for an initial plan or when switching between different phases of a SCORe mission, i.e., from Search to Continuously Observe or from Continuously Observe to React, we can use our solution to let the Multipotent System autonomously adapt the configuration of its agents. Thereby, if there is at least one configuration of the Multipotent System in which it can form an ensemble for handling a plan generated from a SCORe mission, we can find this configuration and let the Multipotent System self-adapt to that configuration. This also holds for whole SCORe missions if we can find such a configuration for any possible plan we can generate from the SCORe mission's description.

While providing such grantees inherently within the Multipotent System is subject to future research, we assume plans and SCORe missions in the following where this is the case, e.g., because the Multipotent System's user took care of it appropriately. Thus, in the following chapter, we can now focus on executing plans in general by assuming that we have a properly configured ensemble at hand for each of these plans.

Chapter 7

Executing Ensemble Programs by Using Self-Organization

Summary. After successfully using the self-organization mechanisms, we propose in Chapter 5 and Chapter 6, we can assume that we now have an appropriately configured ensemble at hand. This assumption holds for any plan we can derive with automated planning from the definitions made by the ensemble programmer using our approach of Maple we introduced in Chapter 4. There, we already described how to derive the respective pieces of information we require for the correct execution of a specific plan. Together, these pieces of information define the control flow of an Ensemble Program, including the required scheduling of instructions to agents that participate in the ensemble $\mathcal{E}^{\rho}$ formed for executing the Ensemble Program. This chapter describes how agents in a Multipotent System that collectively form ensembles for plans can cooperatively execute the associated Ensemble Programs. Thereby, we emphasize two aspects of relevance. First, we provide a solution allowing the agents in an ensemble to coordinate and synchronize the control flow of Ensemble Program in a self-organized fashion. Second, we describe how each agent from the ensemble can correctly execute its respectively scheduled instructions so that the Ensemble Program can make progress using the results of these executions if required. For both aspects, we put particular focus on an algorithmic pattern we found that allows for the generalized execution of different swarm behavior applicable for flying ensembles. We propose our solution for realizing Collective Capabilities we intend to encapsulate such parametrizable, collective behavior for Multipotent System, allowing us to flexibly integrate goal-oriented swarm behavior in Ensemble Program. Our evaluations demonstrate that the pattern we found is appropriate for generalizing swarm behavior based on trajectory modifications of the participating agents. We use this finding to evaluate our concept of Collective Capabilities letting flying ensembles execute different goal-oriented swarm behaviors in exemplary Ensemble Program. We do this by evaluating all features of Maple isolated and integrated. We further evaluate the generality of Collective Capabilities in Multipotent System by exemplary integrating it with Protelis [Pianini et al., 2015], another approach for generalizing collective behavior.

Publication. Contents of this chapter have been published in [Kosak et al., 2019, 2020a; Schörner et al., 2020].

7.1 Introduction

Any plan is only as good as its execution is. According to this statement and after the preparations we made for designing and generating plans in Chapter 4 and preparing our Multipotent System according to this plans in Chapters 5 and 6, the primary goal for the respective ensembles now is to execute those plans according to their specification. Because plans often involve the cooperation of multiple agents, the agents that form the ensemble face two main challenges. On the one hand, the ensemble of agents must be coordinated appropriately. Thus they need to

- select and command the right instructions for all agents in the ensemble,
- collect, synchronize, and process their respective responses, and
- decide for the correct progress in the ensemble program, possibly also involving the user.

On the other hand, agents in the ensemble need to execute instructions commanded to them as defined by the plan. Thus, we require them to correctly,

- determine which capabilities they need to execute with which parameters,
- execute those capabilities accordingly by accessing the required hardware modules in case of physical capabilities,
- appropriately process intermediate results of capability executions in case of virtual capabilities, and
- respond to the coordinating instance at the right time, informing it about the results derived from the commanded instructions.

In this chapter, we now provide a mechanism allowing an ensemble to handle these challenges using measures of self-organization autonomously. Our mechanism can work with any requirements a plan that was generated with our approach of Maple from Chapter 4 can encode. As input, it takes the pieces of information decoded from the respective plan ρ in the way we described it in Section 4.4.5. Together, the coordination info $\mathcal{CI}^\rho$, defining the requirements for the ensemble as a whole, and the respective number of cooperation pattern information $\mathcal{CP}^\rho$, defining the instructions for all agents in the ensemble individually, describe the correct execution scheme of a plan ρ. Besides scheduling the control flow of the Ensemble Program by correctly instructing the agents in the ensemble, the mechanism also triggers manipulations of variables in the world state and the generation of new Ensemble Programs through replanning, if those are commanded in the respective plan.

After scheduling instructions to the agents, the mechanism also synchronizes their executions again if necessary or if explicitly commanded by the plan. Depending on the degree of parallelism, i.e., how many agents are instructed, and the specific parameters of the instructions, i.e., how the encapsulated capabilities should terminate, the synchronization of agents and processing of produced results requires specific handling. This becomes especially relevant if instructions involve Collective Capabilities encapsulating swarm behavior as we provide them in Multipotent System. Thus, we put particular focus on their execution in this chapter.

When designing an HTN as the knowledge base where plans for the ensemble are generated from with automated planning, the ensemble programmer might want to include a respectively parameterized Collective Capability every time it applies to the goal considered by the programmer. That way, the potentially beneficial properties of Collective Capabilities, *scalability and robustness of execution*, can be exploited as often as possible. For enabling that, we pro-

vide the possibility of including goal-oriented swarm behavior encapsulated in such a Collective Capability during the design of HTN (cf. Chapter 4).

For realizing that Collective Capability, we avoid requiring different Collective Capabilities for different swarm behavior that all would require their very own implementation, e.g., on AGENT LAYER and SEMANTIC HARDWARE LAYER of our reference architecture for Multipotent System (cf. Chapter 3), or the respective equivalents of any other MAS/MRS architecture. Instead, we achieved to generalize a whole class of advantageous swarm behavior applicable for flying ensembles into one Collective Capability we can define a general algorithmic pattern for. We call that *Swarm Capability* $c^v_{PROTEASE}$ as it encapsulates an Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE). With c^v_{PROTEASE}, we can command trajectory-modification-based swarm behavior in general, i.e., such swarm behavior that works with modifying the trajectories of entities that can move in 3-dimensional space for achieving an emergent effect. An ensemble programmer influences the micro and macro scale of c^v_{PROTEASE} execution by changing only the parameters when using c^v_{PROTEASE} in different instructions. On the one hand, parameters can influence the behavior on the micro-scale concerning the respective movements of the individual agents. On the other hand, parameters of c^v_{PROTEASE} determine how the surrounding system should exploit the so generated emergent effect on the macro scale of the ensemble. When embedding c^v_{PROTEASE} in an ensemble program, this definition on a macro-scale determines how results from the execution should be derived and processed.

In the following, we illustrate how Ensemble Program execution integrates with the execution of Collective Capabilities. Therefore, we first analyze related work focusing on coordinated execution in MAS/MRS and current developments in applying and generalizing swarm behavior for technical systems in Section 7.2. We then describe the details concerning our concepts in Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE) in Section 7.3. In Section 7.4, we introduce our mechanism for the self-organized execution of Ensemble Program in general. We then illustrate how the execution of Collective Capabilities integrates into this process in Section 7.5. In our evaluations in Section 7.6, we first evaluate the generality of PROTEASE in an isolated MAS simulation, second demonstrate how our engine for executing Ensemble Programs integrates the execution of ensemble level parts and agent level parts in a self-oragnized manner by example, third evaluate our integration of PROTEASE into Ensemble Programs in a goal-oriented fashion in different Ensemble Program as a Collective Capability, and fourth evaluate the extensibility of our concept of Collective Capabilities by integrating it with Protelis [Pianini et al., 2015] which we can also use for exploiting beneficial properties of collective behavior. We conclude this chapter in Section 7.7 by pointing out possible future research directions.

7.2 Related Work

We analyze related work concerning the execution of MAS/MRS missions first before we second have a look at research concerning the usage of goal-oriented swarm behavior in current applications. We investigate solutions focusing on individual swarm behavior for MAS/MRS as well as such aiming at finding generalized solutions for applying swarm behavior.

7.2.1 Coordinated Execution of Multi-Agent/Multi-Robot Missions

Most current approaches focus on single UAV mission control instead of providing solutions for the coordinated execution of multi UAV missions. Examples for such are the APM Mission

Planner [APM, 2020], the QGroundControl [QGroundControl, 2020], the DJI GCS Pro [SZ DJI Technology Co., 2021], or the Paparazzi ground control [Brisset et al., 2006]. While they provide possibilities to the user for useful interactions with the UAV, e.g., monitoring telemetry data during the flight and defining simple missions, they cannot coordinate the cooperation of multiple UAV. Further, the complexity of missions a user can define with the tools mentioned above is relatively low. Usually, they only provide sequential waypoint missions enriched with simple predefined additional commands, e.g., for taking pictures or deploying payloads. Consequently, the architecture and mechanisms for executing those missions are limited, restricting the possibility of applying them to SCORe missions.

An approach for tackling the issue of the lack in multi-UAV support for current tools is the approach of Dousse et al. [2016], aiming at controlling multiple UAV at the same time. There, the authors try to solve the problem with an extension of the QGroundControl [QGroundControl, 2020], which allows the manual control of up to ten UAVs in parallel. While this reduces the maintenance overhead typically required when controlling multiple UAV simultaneously, the approach is still very limited concerning the automation of a mission's execution. The user of the system has to take care manually of every point where synchronization is required. Thus, the approach of Dousse et al. [2016] does not suffice the requirements for controlling ensembles in the way we intend.

Another approach aiming at controlling multi UAV systems is that used by Intel [Intel, 2020] for creating and controlling light shows. The approach allows for the centralized control of the trajectories of several hundred UAVs and their actuators (RGB LEDs). While the resulting light shows demonstrate the possibility of controlling large scale UAV systems, behind the scenes materials [Intel, 2021] give an impression of the very high complexity for planning and controlling such in a centralized fashion. Because with our approach, we aim at the decentralized deployment and execution of missions for versatile use-cases avoiding the necessity for a steady connection to a centralized instance, controlling Multipotent System with the approach of [Intel, 2020] is no option. Further, the technology behind the light shows [Intel, 2021] is proprietary, disallowing its extension and adaption in a way we would require it.

Most frameworks for ensemble programming we analyzed in our related work studies in Chapter 4 also provide mechanisms for the execution of so defined programs.

Lima et al. [2018] come very short when describing their "engine for executing a program" they use in their approach Dolphin. Instead of describing their mechanism for coordinating programs, they focus on the topic of task allocation. This comes because, for every step in their programs, Lima et al. [2018] allocate the respectively relevant tasks to new robots. Thus, there is no need for a complex coordination pattern as Dolphin does not involve "space-time ... task flow" for individual robots in its current state [Lima et al., 2018], i.e., robots do not need to be aware of their respective current state and the progress in a mission. Further, Dolphin does not provide the possibility for executing Collective Capabilities of any kind and thus lacks an execution engine for such.

Koutsoubelias and Lalis [2016] use a centralized instance for controlling the execution of programs created with their approach TeCola. Because robots in TeCola are looked at as services that this central instance can address, there is less effort to define measures for coordinating program control flow and data transmission. While we let an ensemble executing an Ensemble Program designed in Maple execute these functionalities completely autonomous, programs in TeCola bundle them centrally. Thus, there is no need for distributing the information necessary for controlling the execution of TeCola programs. Because TeCola does not support a concept comparable to Collective Capabilities we provide in Multipotent System to

encapsulate swarm behavior, TeCola does not support an engine for executing such.

Mottola et al. [2014] provide two different models for executing missions deployed to multi UAV systems. In a centralized execution model, no data is stored on the individual UAV, i.e., the central instance manages all program control flow and all data generated during execution. The multi UAV program is replicated for every individual UAV and constantly synchronized with frequent messages in their distributed execution model. In both cases, the coordination of UAV in [Mottola et al., 2014] is restricted to planning individual trajectories and avoiding collisions while executing them. Because the possible actions UAV can execute during a mission combined with the lack in supporting a concept comparable to Collective Capabilities, there is no need for complex coordination of execution as we require and provide it in our approach.

Another approach for executing multi robot programs is that of using the Robot Operating System (ROS) middleware for robot applications [Quigley et al., 2009]. With a publish/subscribe messaging interface enabling the communication of components in a robot program, ROS provides the possibility to realize the coordination of MAS/MRS missions. Especially with its extension, ROS2, this distribution of execution is in focus. In [Schörner et al., 2020], we make use of the possibilities ROS2 and accompanying frameworks deliver for realizing a mission definition and execution framework. While we investigated the problem of coordinately controlling multiple UAV focusing on the actions of individual UAV and their synchronization, we did not investigate the ensemble level. Thus, with our approach in [Schörner et al., 2020], we could not yet instruct and control the execution of Collective Capabilities that, e.g., can include swarm behavior.

Gutmann and Rinner [2021] make also use of the possibilities that ROS provides for the distributed execution of multi UAV missions. Therefore, the authors provide their idea of a mission execution stack working with a layered execution architecture with a similar division of concepts and functionality we propose in our reference architecture for Multipotent System. In contrast to our approach, Gutmann and Rinner [2021] only distribute the lowest layer, i.e., the hardware layer, to different devices. Thus, because the execution engine for missions is still deployed on a central controlling instance, there is no need to distribute necessary information on program control flow and data flow to the individual devices of the respective system. Because their approach is still in the conceptual phase, they postpone "finalizing the execution architecture development" to future work. Gutmann and Rinner [2021] do not involve the possibility to include collective behavior as we do with Collective Capabilities encapsulating goal-oriented swarm behavior in their approach, nor do they plan for such in future work.

7.2.2 Swarm Behaviors and Approaches for Their Generalization

The literature on swarm behavior, swarm algorithms, or swarm intelligence is manifold. When swarm behavior should be exploited in a real-world application, there are two directions researchers currently follow. The first direction is to focus on one specific behavior found in nature that gets analyzed and migrated to technical systems. Examples for that direction are manifold. Thus we only can give an excerpt of research relevant in this section. To achieve a collective transport of an object, the authors in [Dorigo et al., 2004; Mondada et al., 2005] developed a specialized controller by using an evolutionary algorithm for mobile ground robots. While they achieve the desired effect, they suffer from the evolutionary algorithms inherent properties of high specialization and the lack of generality: The generated controller cannot be used in any other use case. To achieve a close-to equal distribution of swarm entities in a given area, e.g., for Distributed Surveillance, Ma and Yang [2007] adapt a potential-field-based

deployment algorithm. Unfortunately, the algorithm thus can only be used for exactly that use case. While Li et al. [2009] propose that they can adapt their swarm approach for Distributed Surveillance also to achieve flocking and obstacle avoidance, they, unfortunately, do not further investigate in this direction. In our opinion, this is a step in the right direction to generate a general pattern for achieving swarm behavior which we try to make with our approach. In [Sánchez-García et al., 2019] the authors adapt the Particle Swarm Optimization (PSO) for the use of UAV in disaster scenarios to explore an area and detect victims. While the authors can adapt parameters to achieve different goals, the approach is still limited to that narrowly defined area and cannot easily be extended. With an adapted flocking algorithm based on the approach of Reynolds [1987], the authors in [Vásárhelyi et al., 2014] demonstrate how UAVs can achieve swarm behavior that is very close to that of natural swarms. Unfortunately, the implementation is very specific and can solely achieve this concrete swarm behavior.

The second direction researchers follow is to abstract from specific applications and use cases and developing a general framework for collective behavior that can be programmed or parametrized in different ways. Protelis [Pianini et al., 2015] is one approach we also categorize as such. The authors center it around the idea of abstracting entities in a collective system as a point in a high-dimensional vector field. Programming of the collective happens by performing operations on that field. By using implicit communication between entities, the programmer can achieve that changes performed in these fields are distributed within the collective. While a user can exploit this behavior to implement complex collective on an abstract level, it is not easy to achieve goal-oriented swarm behavior for complex mobile robot tasks solely with Protelis. Its lack of general hardware integration and a general task concept enabling interconnecting tasks with task orchestration makes using Protelis alone insufficient for complex goal-oriented robot collaboration like we require in SCORe missions. Because Protelis offers some advantageous properties like self-stabilization of programs written with it, we provide a solution for integrating it into Maple, allowing for its execution in Ensemble Programs and evaluating our concept of Collective Capabilities (cf. Sections 7.5.4 and 7.6.2.5).

Another programming language aiming at collective systems is Buzz Pinciroli and Beltrame [2016]. In comparison to Protelis, the authors of Buzz directly aim at integrating their programming language within robot operating systems. They provide swarm primitives for achieving a specific desired collective behavior each. Unfortunately, Buzz also lacks a concept for goal-oriented task orchestration. Further and like for using Protelis, a user of Buzz currently requires a system specifically designed for the respective programming language.

Ashley-Rollman et al. [2007] also focus on providing abstractions for controlling ensembles with their approach Meld. Having a look at the execution model of Meld shows that assumptions made there are far away from the reality one is confronted with when programming flying ensembles. Calculations for different ensemble behaviors highlighted in [Ashley-Rollman et al., 2007] require immense time (e.g., up to 120 minutes per node of the system for solving a routing problem). Further, robots in the focus of [Ashley-Rollman et al., 2007] are so limited in their capabilities that thinking about programming Ensemble Programs in the context of SCORe missions is not realistic. Further, all examples using Meld currently are simulation-based only. Thus, while Meld offers an interesting new way for programming ensembles (cf. our analysis in Chapter 4), its execution model is far too limited compared to our approach.

Dedousis and Kalogeraki [2018] propose swarm primitives for controlling multi UAV systems with their approach PaROS. While they aim at realizing swarm behavior for specific tasks in these primitives, they do not focus on generalizing swarm behavior as a whole. Further, PaROS controls the swarm from a centralized instance in the system. Moreover, PaROS

does not support task orchestration, i.e., results of primitives cannot be combined to complex behavior. Thus, there is no need for complex scheduling the control flow as we require it in MAPLE.

Varughese et al. [2020] investigate the possibilities of generalizing swarm behavior. They provide a design paradigm for realizing certain classes of swarm behavior called *primitives* focusing on as-low-as-possible communication requirements. This comes as the approach of Varughese et al. [2020] aims at realizing the primitives for minimalistic and completely decentralized working robots. For such, Varughese et al. [2020] propose that combinations of primitives can achieve complex behavior like collective transport in a 2-dimensional environment. Because the paradigm they present is based on cyclic synchronization messages propagating through the swarm realizing one of the primitives, execution times for stabilizing in the desired state are very high (e.g., distributed localization, one of the primitives, requires 1400 seconds). While the paradigm, in general, provides the intriguing possibility of transporting life-like swarm behavior to a technical system having in mind to generalize similar behavior, it seems inefficient for applying it to flying ensembles. There, we require to step back a little from the idea of complete decentralization for gaining performance in the execution.

7.3 A General Algorithmic Pattern for Trajectory-Based Swarm Behaviors (PROTEASE)

We have seen throughout the last chapters that the use of multi UAV systems, ensembles, and swarms of them can be beneficial in many situations in our daily life. This statement is validated by the multitude of different applications for ensembles that emerged during the past decade, using the benefits collective behavior can deliver, e.g., with emergent effects achieved by swarm behavior. Unfortunately, the current trend is that every new application also requires a new software approach for its realization [Dedousis and Kalogeraki, 2018]. While these specialized approaches show beneficial results for their dedicated applications, e.g., using collective swarm behavior for searching [Zhang et al., 2015], or Distributed Surveillance [Ma and Yang, 2007; Li et al., 2009] among many others, users can find it hard to adapt them and profit from previous developments in (even only slightly) different use cases.

To come by this issue, we propose to make use of a *common pattern* instead that can express the swarm behavior of a particular class in general. Developers of multi-robot systems can implement such a pattern once at design time and parametrize it differently at run-time to achieve specific emergent effects. We identified such an Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE).[1] Therewith, we aim at generalizing different functionality researchers frequently use for implementing different types of swarm behavior in swarm robotic systems. While producing a different emergent effect each, we can see that swarm algorithms like the particle swarm optimization algorithm [Zhang et al., 2015], the commonly known flocking behavior initially analyzed in [Reynolds, 1987], shaping and formation algorithms [Rubenstein et al., 2014], and distribution algorithms [Ma and Yang, 2007; Li et al., 2009] make use of the same set of local actions: Measuring one or multiple specific parameters, communicating with neighbors in the swarm, and modifying the movement vector of the robot. For example, to realize flocking behavior following Reynolds [1987], each "boid" requires to execute certain functionality in an appropriate combination, i.e., performs position and velocity

[1] "... proteases are key regulators of a striking variety of biological processes ... they regulate different processes in response to developmental and environmental cues" [Van der Hoorn, 2008].

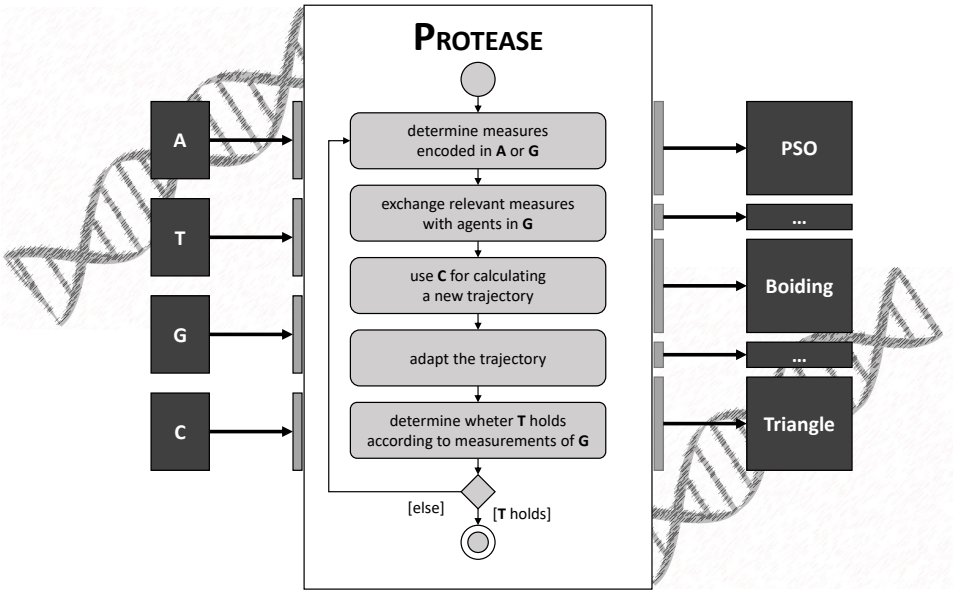


Figure 7.1: The Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE) every agent participating in a swarm needs to execute.

measurements, exchanges resulting values with swarm members, and adapts its movement vector accordingly, which then results in the collective emergent effect of the individuals forming a flock as an ensemble.

Regarding this finding, we define the parameters for PROTEASE necessary to produce a wide range of swarm behavior in the following. Thereby, we have in mind to enable the integration of PROTEASE into an approach for task orchestration for flying ensembles like we provide in MAPLE. The parameters forming the behavior of PROTEASE thus are

- *A:* Aggregation Function of PROTEASE ($\text{AGG}_{\text{PROTEASE}}$) enabling the ensemble executing PROTEASE to determine the collective result of its execution,
- *T:* Termination Function of PROTEASE ($\text{TERM}_{\text{PROTEASE}}$) for letting an ensemble executing PROTEASE determine that it has reached the desired goal,
- *G:* Group Function of PROTEASE ($\text{GROUP}_{\text{PROTEASE}}$) defining the set of agents in the neighborhood to exchange information with during the execution of PROTEASE, and
- *C:* Calculation Function of PROTEASE ($\text{CALC}_{\text{PROTEASE}}$) determining how information generated within the group of agents in the ensemble executing PROTEASE should be derived and how they should be transformed into a modification of the respective agents trajectory to move with.

Examples for swarm behavior we can express by executing PROTEASE with different parameters ATGC[2], i.e., PROTEASE(ATGC), are

- *Search* the spot of highest concentration of a continuously distributed parameter of interest, e.g., by adapting the concepts of Particle Swarm Optimization (PSO) surveyed in [Zhang et al., 2015],

[2]In case of DNA, "each nucleic acid contains four ... nitrogen-containing bases: adenine (A), guanine (G), cytosine (C), [and] thymine (T)" [Roberts, 2020]. "Nucleic acids are the main information-carrying molecules ... they determine the inherited characteristics of every living thing" [Roberts, 2020].

  - $\text{AGG}_{\text{PROTEASE}}$: the centroid regarding the positions of all agents in $\text{GROUP}_{\text{PROTEASE}}$
  - $\text{TERM}_{\text{PROTEASE}}$: the geometric diameter regarding the positions of agents described by $\text{GROUP}_{\text{PROTEASE}}$ is below a threshold
  - $\text{GROUP}_{\text{PROTEASE}}$: the group of agents executing the PSO
  - $\text{CALC}_{\text{PROTEASE}}$: the local rules of PSO involving a specific parameter of interest
- *Continuously Observe* a certain area of interest using, e.g., a triangle-formation algorithm like proposed by [Li et al., 2009],
  - $\text{AGG}_{\text{PROTEASE}}$: the position of an event that happened during the observation
  - $\text{TERM}_{\text{PROTEASE}}$: the destination of the event resulting from $\text{AGG}_{\text{PROTEASE}}$
  - $\text{GROUP}_{\text{PROTEASE}}$: the group of agents executing the triangle formation algorithm
  - $\text{CALC}_{\text{PROTEASE}}$: the specific local rules of the triangle formation swarm behavior
- ˚ to unforeseen situations by gathering agents at a rendezvouz position using, e.g., the swarms centroid as goal destination.
  - $\text{AGG}_{\text{PROTEASE}}$: the finally determined rendezvouz position
  - $\text{TERM}_{\text{PROTEASE}}$: the geometric diameter regarding the positions of agents defined by $\text{GROUP}_{\text{PROTEASE}}$ is below a threshold
  - $\text{GROUP}_{\text{PROTEASE}}$: the group of agents executing the gathering swarm behavior
  - $\text{CALC}_{\text{PROTEASE}}$: the specific local rules directing an agent to the rendezvouz position

Besides these examples, we can express many other swarm behavior with parameters ATGC and execute them with PROTEASE. We investigate in such in our evaluations in Section 7.6.1.2.

To apply to our general pattern of PROTEASE, swarm behavior must comply with some key elements. In its core form, the swarm behavior must work without the need for marking the environment in a stigmergic style like it is common in ant-colony-optimization-based swarm algorithms [Bianchi et al., 2002]. Because we intend PROTEASE to work with flying ensembles, it is not easy to realize stigmergy. Obviously, we can only achieve such with a digital representation of the environment where all agents executing PROTEASE need to have access to simultaneously. While we are currently investigating the possibility of realizing such, we want to abstract from such swarm behavior in this thesis. Further, we focus on such swarm behavior that produces its emergent effect as the result of modifying the participating agents' trajectories. Thus, we cannot guarantee that swarm behavior basing its emergent effect on other agent functionality can also be expressed with PROTEASE.

Further, to make PROTEASE work within flying ensembles, we require that every agent can communicate with any other agent in the respective group within the ensemble executing it. This is necessary to realize certain types of swarm behavior, e.g., PSO following the descriptions of Zhang et al. [2015]. Further, we cannot assume local sensors for all spatially distributed relevant values, e.g., precisely and fast enough measurement of the other agents' positions. Moreover, as the core functionality of PROTEASE is to modify the participating agents' trajectories in 3-dimensional space, we require agents to be able to move respectively.

7.4 Coordinated Execution of Ensemble Programs

We now introduce our mechanism for executing Ensemble Program by describing the necessary logic to be executed by all agents involved. For a specific plan, this covers all agents $\alpha \in \mathcal{E}^{\rho}$ that

Algorithm 4 EPU PROGRAM ($\mathcal{CP}^{\rho}$)

```
1: I_PC ← ∅
2: procedure AS:COORD(PC, R_{E_PFC}, WS)
3:   STOP(C_RUNNING)
4:   C_RUNNING ← ∅
5:   s_EPU ← ⊤
6:   e_EPU ← ⊥
7:   R_EPU ← [ ]
8:   I_EPU ← CP^ρ(PC)
9:   for INSTR ∈ I_EPU do parallel
10:    c ← INSTR.c
11:    p ← INSTR.p
12:    s ← INSTR.s
13:    e ← INSTR.e
14:    UPDATE(p, R_{E_PFC}, WS)
15:    s_EPU ← s_EPU ∨ s
16:    e_EPU ← e_EPU ∧ e
17:    if s then
18:      R_EPU[INSTR] ← EXEC(c, p)
19:    else
20:      C_RUNNING ∪ c
21:      EX(c, p)^↑
22:  join
23:  if s_EPU then
24:    STOP(C_s)
25:  ℂALLER.SYNC(s_EPU, e_EPU, R_EPU)
```

Figure 7.2: EPU program every agent implements to realize the agent level part of an Ensemble Program.

execute instructions and the one specific agent coordinating the control flow of the Ensemble Program, i.e., the realization of the agent level part in Ensemble Processing Units EPU ∈ EPU and the realization of the ensemble level part of the Ensemble Program in a Program Flow Controlling instance PFC we described in Section 4.2.

To decode the requirements for an ensemble $\mathcal{E}^{\rho}$ from a plan ρ, we can generate different pieces of information as we describe them in Section 4.4.5. Together, one coordination information $\mathcal{CI}^{\rho}$ for the agent realizing the PFC of the Ensemble Program and a respective number of cooperation pattern information $\mathcal{CP}^{\rho}$ for the agents realizing the EPU ∈ EPU ensure the correct execution of a plan ρ when used as input for our mechanism. The mechanism we propose can manage the control flow of the Ensemble Program on the ensemble level and the individual agents' level using the pieces of information mentioned above. By scheduling the instructions to the agents in the ensemble in the order defined by $\mathcal{CI}^{\rho}$, it can ensure the right order of execution in sequential, parallel, conditional, repeated, and concurrent executions. Knowing the single instructions' properties concerning their termination criteria and return values, the mechanism can decide when to synchronize the program flow that the agents of the ensemble distributedly executed and when the Ensemble Program requires additional input from the user to make progress. Working with the information from $\mathcal{CI}^{\rho}$, our mechanism allows triggering world state modifications and replannings when necessary. We use the notation we introduced in Section 4.2 for describing the programs in this section.

7.4.1 The Agent Level Part of an Ensemble Program

We first focus the agents' level of our mechanism, i.e., the agent level part, describing how instructions encoded in $\mathcal{CP}^{\rho}$ can be executed individually by the respectively instructed agent.

Agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$ realize the concept of EPU $\in$ EPU in our system. Having ensured by task allocation during Ensemble Formation (cf. Chapter 5), we know that each agent provides all capabilities necessary for executing the instructions contained in its respective $\mathcal{CP}^{\rho}$ associated to the task it finally got assigned. With that $\mathcal{CP}^{\rho}$, each $\alpha \in \mathcal{E}^{\rho}$ further gets the information about how to execute a capability c, i.e., retrieves functional parameters p and non-functional parameters s, and e for each capability. We presented the datatype of such $\mathcal{CP}^{\rho}$ in Table 4.2, pointing out the information contained there. For the correct execution of Ensemble Programs defined with our approach MAPLE, each agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ thus implements the behavior defined in Algorithm 4 that is instantiated with the respective $\mathcal{CP}^{\rho}$ relevant for the respective plan ρ.

When an agent α receives a coordination message (cf. Line 2 in Algorithm 4), it needs to execute all instructions INSTR $\in \mathcal{I}_{\text{EPU}}$ in parallel that are referenced by the transmitted PC in its $\mathcal{CP}^{\rho}$ (cf. Line 9 in Algorithm 4). To derive the relevant set of instruction $\mathcal{I}_{\text{EPU}}$, the agent can access $\mathcal{CP}^{\rho}$ with the transmitted program counter PC (cf. Line 8 in Algorithm 4). The agent then needs to execute all instructions INSTR $\in \mathcal{I}_{\text{EPU}}$ with the respectively transmited parameters p, s, and e. The agent can derive this information by accessing a respective field defined in the complex data type INSTR where the ensemble programmer set the value during programming, e.g., retrieve the capability c addressed in the instruction with INSTR.c or the finishing type of the instruction with INSTR.s (cf. Line 10 to 13 in Algorithm 4). Before actually starting the respective capability's execution, the agent first updates the capability's functional parameter p with UPDATE$(p, R_{\mathcal{E}_{\text{PFC}}}, \text{WS})$, if p contains not only constant values but also variables (cf. Section 4.4.2). It does this using the current values of variables from the shared storage WS and such generated by the ensemble during previous executions ($R_{\mathcal{E}_{\text{PFC}}}$) if necessary in Line 14 of Algorithm 4.

For the further execution of instructions, we make some assumption for such encoding the execution of Collective Capabilities and such encoding the execution of other capabilities. While the EPU program makes no difference between those types of instructions, and we support to execute other instructions in addition to a Collective Capability in the same $\mathcal{I}_{\text{EPU}}$ in principle, we currently recommend avoiding such combinations regarding possible conflicts during their execution. Such can be caused, e.g., by including instructions encoding multiple Collective Capabilities for the same agent, whose effect we no longer can estimate. Thus, we assume that Collective Capabilities are executed exclusively in a specific PC. If $\mathcal{I}_{\text{PC}}$ includes an instruction addressing a Collective Capability like c^{v}_{PROTEASE}, i.e., $c = c^{v}_{\text{PROTEASE}}$, like for every other capability, the local rule set for executing the respective swarm algorithm is encoded in an extra piece of code. Because Collective Capabilities compared to other virtual capabilities are executed collectively and thus show some specific properties during execution, we describe them in detail in the next Section 7.5.

According to the value of s, the respective capability c included in INSTR is executed self-finishing ($s = \top$) or non-self-finishing ($s = \bot$), using the functional parameter encoded in INSTR (cf. Line 17 – Line 21 in Algorithm 4). If INSTR.$s = \top$, the agent can first store the result locally (cf. Line 18 in Algorithm 4, if, e.g., $c = c^{p}_{\text{MV-POS}}$) and second, after all other capability executions are finished, synchronize the results R_{EPU} with the coordinating instance (cf. Line 25 in Algorithm 4). If INSTR.$s = \bot$, the agent can only start the execution of c and wait for some signal to terminate c again. Therefore, the agent first remembers the respective capability by adding it to a set C_{RUNNING} and then starts the capability's execution (cf. Line 20 and 21 in Algorithm 4). The signal for terminating those capabilities $c \in C_{\text{RUNNING}}$ an agent α_i is executing can come from different levels of coordination.

- The signal can come from the execution of other instructions INSTR $\in \mathcal{I}_{\text{PC}}$ that agent α_i can finish on their own, i.e., such that have INSTR.s set to $\top$. Then, the agent can finally set s_{EPU} to $\top$ (cf. Line 24 in Algorithm 4).
- The signal must come from outside agent α_i executing the EPU program but can come from within the ensemble if other agents α_j with $\alpha_{i \neq j}$ and $\alpha_j \in \mathcal{E}^\rho$ can terminate their executions while the α_i cannot do that locally, i.e., $\forall$INSTR $\in \mathcal{I}_{\text{EPU}} : s = \bot$. Then, still running capabilities can be terminated after receiving a subsequent coordination signal triggered by the algorithm we describe in Section 7.4.2 (cf. Line 3 in Algorithm 4).
- The signal must com from outside the agent α_i and the ensemble $\mathcal{E}^\rho$, i.e., from the user, if neither agent α_i itself nor the whole ensemble, i.e., no agent $\alpha_j \in \mathcal{E}^\rho$ can produce the signal to finish or, the user explicitly commanded to be involved in the progress by setting INSTR.e to $\top$ (cf. Line 16 in Algorithm 4).

If the agent cannot determine to terminate the execution of capabilities started with $s = \top$, it requires to wait for external control received with the next PC. To inform the controlling instance, i.e., the $\mathbb{C}$ALLER of COORD(PC), appropriately, the agent thus needs to evaluate whether such external coordination is required (cf. Line 15 and 16) and include this information in its response message SYNC($s_{\text{EPU}}, e_{\text{EPU}}, R_{\text{EPU}}$) (cf. Line 25).

7.4.2 The Ensemble Level Part of an Ensemble Program

In each ensemble, we require one agent to realize the functionality of a Program Flow Controlling instance PFC, coordinating the control and data flow within the ensemble. This is especially relevant for deciding to make progress in an Ensemble Program when agents in the ensemble execute instructions in parallel, i.e., within the same PC of the Ensemble Program. The PFC provides an interface to the user allowing to influence the execution of Ensemble Program. Further, only a PFC can collect all information necessary for deciding on the progress of conditional control flow by aggregating and evaluating the required information possibly generated distributively by different agents in the ensemble. Moreover, the PFC can use such information to access and modify variables stored in the world state and trigger replanning, i.e., the autonomous generation of new Ensemble Program. To decide on the correct action to perform on ensemble level, the agent realizing the functionality of PFC uses the $\mathcal{CI}^\rho$ we generate for a specific plan ρ as input. For supporting all controlling functions, we propose that each agent that realizes the functionality of PFC in an ensemble implements the PFC program depicted in Algorithm 5. Algorithm 5 thus is the *ensemble complement*, i.e., the ensemble level part of an Ensemble Program, for the respective EPU programs depicted in Algorithm 4. As the respective complement to $\mathcal{CP}^\rho$ we use as input for EPU programs, we use the $\mathcal{CI}^\rho$ as input for the PFC program. This $\mathcal{CI}^\rho$ includes information on all participating agents in the ensemble $\mathcal{E}^\rho$, as well as control-flow and data-flow information referenced by unique PCs. We showed the datatype of $\mathcal{CI}^\rho$ in Table 4.1, pointing out possible information contained there. While we asssumed that $\mathcal{E}^\rho$ in Table 4.1 was filled with place-holding Planning-Agent in Chapter 4, we now have filled those placeholders with actual agents $\alpha \in \mathcal{A}_{\mathcal{MS}}$.

7.4.2.1 Coordinating the Control-Flow

The foremost functionality of an PFC program is deciding on the correct action to perform for each PC referenced in the $\mathcal{CI}^\rho$. Each PC in a $\mathcal{CI}^\rho$ can reference different required actions to be executed by the PFC program, determined by different TYPES and enriched with additional

Algorithm 5 PFC PROGRAM

```
1:  user; PC ← 0
2:  procedure AS:START(CI^ρ)
3:    E_PFC ← CI^ρ.E^ρ
4:    EXECUTE()
5:  procedure SELECTPC
6:    SUCC-MAP ← CI^ρ.PC_SUCC-MAP[PC]
7:    if PC = 0 then
8:      PC ← 1
9:    else
10:     PC ← −1
11:     for CONDITION ∈ PC_succ.keys do
12:       t ← EVALUATE(CONDITION, WS, R_E^ρ)
13:       if t then
14:         PC ← PC_succ[CONDITION]
15: procedure COORDINATE
16:   for α ∈ E_PFC do parallel
17:     α.COORD(PC, R_E_PFC, WS)^↑
18:   join
19: procedure AS:SYNC(s_α, e_α, R_α)
20:   E_PFCwait ← E_PFCwait ∪ ℂALLER
21:   R_E_PFC[α] ← R_α
22:   e ← (e ∧ e_α)
23:   s ← (s ∨ s_α)
24:   if E_PFC ⊂ E^ρ_wait then
25:     if e ∨ ¬s then
26:       user.REQUESTCOORDINATION()^↓
27:     EXECUTE()
28: procedure EXECUTE
29:   SELECTPC()
30:   if PC = −1 then
31:     TERMINATE()
32:   TYPE ← CI^ρ.PC_TYPE[PC]
33:   EXP ← CI^ρ.PC_EXP[PC]
34:   switch TYPE do
35:     case EX
36:       E^ρ_w ← ∅; e ← ⊥; s ← ⊤
37:       u ← ⊥; u_r ← ⊥
38:       COORDINATE()
39:     case STORE
40:       UPDATEWS(EXP)
41:       EXECUTE()
42:     case PLAN
43:       plans ← CREATEPLAN(EXP)
44:       BROAD(plans)
45:       EXECUTE()
46:     case SPLIT
47:       plans ← EXP
48:       BROAD(plans)
49:       EXECUTE()
50:     case FINISH
51:       COORDINATE()
```

Figure 7.3: PFC program every agent implements to realize the ensemble level part of an Ensemble Program.

information included in EXP, if necessary (cf. Chapter 4 for possibilities to define them). After starting the PFC program by calling the procedure AS:START($\mathcal{CI}^\rho$) we intend to be an asynchronous service, we retrieve the set of agents to be coordinated by the PFC program from the $\mathcal{CI}^\rho$ and start the program's execution by calling the internal procedure EXECUTE() (cf. Line 2 to 4 in Algorithm 5). The execution of the PFC program then can be described as a state machine that first determines the relevant PC in SELECTPC() (cf. Line 29 in Algorithm 5), retrieves the type TYPE and expression EXP of this PC from the $\mathcal{CI}^\rho$ (cf. Line 32 and 33 in Algorithm 5) and then executes the associated action (cf. Line 34 to Line 51 in Algorithm 5).

- If the type of PC is EX, we require all agents in $\mathcal{E}^\rho$ to execute the instructions referenced by the current PC in their respective $\mathcal{CP}^\rho$ using their EPU programs (Line 35 to 38 in Algorithm 5). Because this coordination potentially involves asynchronous and parallel program control flow triggered by addressing all agents $\alpha \in \mathcal{E}^\rho$ when calling COORDINATE(), we need to carefully handle the further execution of the PFC program. Thus, we do not directly call EXECUTE() after starting the parallel execution but describe how we progress in the Ensemble Program in that case in Section 7.4.2.3. For a PC of type EX, there is no content in EXP because all execution of instructions takes place in the agents' EPU programs.
- If the type of PC is STORE, we require the PFC to perform a modification to variables in the shared storage WS (cf. Line 39 to 41 in Algorithm 5). For a PC of type STORE, EXP contains statements commanding how to modify values of variables in the shared storage WS. After storing, the control flow continues by calling EXECUTE().
- If the type of PC is PLAN, we require the PFC to generate a new Ensemble Program by planning with the HTN (cf. Line 42 to 45 in Algorithm 5). For each PC of type PLAN, EXP references a Complex-Node (CN) in the HTN where replanning should be started for generating a new Ensemble Program. We described the details for generating new plans in Section 4.4.6. After finishing planning, we can broadcast the new plan within the system and continue executing the PFC program by calling EXECUTE().
- If the type of PC is SPLIT, we require the PFC to broadcast concurrent plans to the system so that other ensembles can start their execution. For each PC of type SPLIT, EXP gives a reference to the concurrent plans to be broadcasted (cf. Line 46 to 49 in Algorithm 5). After broadcasting the concurrent plans, we can continue the PFC program by calling EXECUTE().
- If the type of PC is FINISH, we require to inform all agents in $\mathcal{E}^\rho$ to stop executing capabilities that they are possibly still running and dissolve the current ensemble (cf. Line 51 in Algorithm 5). For a PC of type FINISH, no instruction is needed as this functionality does not differ in different Ensemble Program. After calling COORDINATE() the last time for that $\mathcal{CI}^\rho$, we return to EXECUTE() a last time after finishing the coordination routine we describe in Section 7.4.2.3 with a PC set back to -1 in SELECTPC(), causing the PFC program to terminate its current execution (cf. Line 30 and 31 in Algorithm 5).

For each PC, an $\mathcal{CI}^\rho$ encodes a successor map mapping conditions to PC. Conditions in that map need to be excluding each other and are formulated using variables defined in the shared storage WS, so that the agent executing the PFC program can deterministically determine the next PC in the Ensemble Program. We describe the process of selecting the succeeding PC in cf. Section 7.4.2.2. The PFC program finishes its execution, if the PC is set to -1 by this routine after all its instructions have finally finished (Line 30 and 31 in Algorithm 5).

7.4.2.2 Determining the Succeeding Program Counter

We can determine the next program counter in an PFC program by using the conditions the user defined for each PC referencing possible follow-up PC we encode in the successor map $\text{PC}_{\text{SUCC-MAP}}$ contained in $\mathcal{CI}^{\rho}$ (cf. Table 4.1). We do this by executing the procedure SELECTPC (cf. Line 5 to Line 14 in Algorithm 5) which we describe in the following. Because we do not support indeterminism when starting the execution of an PFC program, we set PC to 1 from the initial value of 0 in the first call of SELECTPC() (cf. Line 8 in Algorithm 5). For each subsequent call of SELECTPC() during the execution of the PFC program, we then decide on the next PC by evaluating the conditions given as keys in the respective map PC_{succ} (cf. Table 4.1) referencing a new PC for each key. Because we require these CONDITIONs to be excluding each other and we require one default successor for each PC if none of the listed conditions hold (i.e., an *else* case), the result of this evaluation is deterministic. For evaluating conditions with EVALUATE(CONDITION, WS, $R_{\mathcal{E}^{\rho}}$) we let the PFC program take into account the current values of variables in the shared storage WS as well as the ensemble-internal results $R_{\mathcal{E}^{\rho}}$ of previous executions performed in the program (cf. Line 39 in Algorithm 5). When starting SELECTPC() for the last time, i.e., the type of the current PC is FINISH, we have no entry in PC_{succ} for potential successors. Thus, we leave the value of PC set to -1 (cf. Line 10 in Algorithm 5), causing the program to terminate subsequently (cf. Line 31 in Algorithm 5).

7.4.2.3 Coordinating Parallelism

The PFC program (cf. Algorithm 5) interacts with EPU programs (cf. Algorithm 4) by calling the service AS:COORD(PC, $R_{\mathcal{E}_{\text{PFC}}}$, WS) asynchronously (cf. Line 17 in Algorithm 5 and Line 2 in Algorithm 4) and receiving SYNC($s_{\alpha}, e_{\alpha}, R_{\alpha}$) messages as response (cf. Line 19 in Algorithm 5 and Line 25 in Algorithm 5). Using such asynchronous and parallel program control flow, we can achieve that Ensemble Program are executed the way the programmer intends them, i.e., realize logical and physical parallel execution of instructions with any possible functional and non-functional parameters. Each call of AS:COORD(PC, $R_{\mathcal{E}_{\text{PFC}}}$, WS) triggers a SYNC($s_{\alpha}, e_{\alpha}, R_{\alpha}$) response from the respective agent executing the EPU program, independent of which agent $\alpha \in \mathcal{E}^{\rho}$ the PFC program addresses and how the referenced instructions are parametrized. Depending on the parameters of instructions an agent needs to execute for the transmitted PC according to its individual $\mathcal{CP}^{\rho}$, the point in time and the information contained in the response can differ. If the instructions the agent should execute include such that can self-finish their execution, the agent can respond to the PFC program after those have finished their execution. Suppose there were only non-self-finishing capabilities included in the instructions the agent had to execute, or the PC referenced no instructions at all. In that case, the agent can instantly send its response back to the PFC program after executing all capabilities encapsulated in the instructions. This guarantees the synchronization of the ensemble and enables the starting and stopping of physically parallel executions, i.e., the PFC program can collect all responses of agents (cf. Line 20 in Algorithm 5) and determine whether all ensemble members are synchronized or not (cf. Line 24 in Algorithm 5). Depending on the respective non-functional parameters s and e capabilities were started by the agent, the agent's response then informs the PFC program about its current state. When any agent's response calls for user coordination (cf. Line 22 in Algorithm 5), i.e., any of the instructions that agent had to execute for this PC was started with e set to $\top$ (indicated with $e_{\alpha} = \top$ in the response), or all agents require external coordination for terminating the executing of their non-self-finishing capabilities (cf.

Line 23 in Algorithm 5), i.e., if all instructions the agent had to execute for this PC were started with s set to $\perp$ (indicated with $s_\alpha = \perp$ in the response), the PFC program needs to involve the user for making progress (cf. Line 26 in Algorithm 5 - we wait for the user with a synchronous call of REQUESTCOORDINATION()$^{\downarrow}$). Otherwise, either the agent itself already achieved to stop its non-self-finishing capabilities internally or the PFC program can achieve such with the next call of AS:COORD(PC, $R_{\mathcal{E}_{\text{PFC}}}$, WS) triggered during the subsequent execution of EXECUTE() (cf. Line 27 in Algorithm 5). Because also the last PC of an Ensemble Program can instruct agents to execute non-self-finishing capabilities, we require to send a final call of AS:COORD(PC, $R_{\mathcal{E}_{\text{PFC}}}$, WS) to all agents when finishing the Ensemble Program execution (cf. Line 51 in Algorithm 5).

7.5 Executing Collective Capabilities in Ensemble Programs

After describing the process of coordinately executing Ensemble Programs involving Non-Collective Capabilities, we now put focus on executing such Collective Capabilities. Therefore, we describe their functionality and how they integrate within the coordinated execution between EPU programs and PFC programs we described in Section 7.4.

Collective Capabilities show two main points of difference when compared to many other capabilities. First of all, Collective Capabilities are virtual capabilities that combine the execution of multiple other capabilities in a complex fashion. Second, Collective Capabilities rely on the direct exchange of information between agents executing them, i.e., such agents that are part of the respective collective. This is an urgent criterion for many collective behaviors like swarm behavior we intend and design Collective Capabilities for, but also for any other aggregate/swarm/ensemble programming approach we can find in the literature (cf. Section 7.2.2).

7.5.1 General Design of Collective Capabilities

Because the execution of each Collective Capability c^v_{COLL} requires the cooperation within the collective $\mathcal{E}_{\text{COLL}}$ executing it, we allow for every agent α_i executing a specific Collective Capability to directly exchange information with other agents $\alpha_{j\neq i}$ within the same collective $\mathcal{E}_{\text{COLL}}$ that are executing the same instance of c^v_{COLL}. We therefore separate each Collective Capability in an active part $c^{v:\text{ACT}}_{\text{COLL}}$ and a passive part $c^{v:\text{PAS}}_{\text{COLL}}$. While the active part differs for all Collective Capabilities, we can define the passive part as a procedure RECEIVE($c^v_{\text{COLL}}, \mathcal{V}_{\alpha_i}$) in general. Each agent uses this passive part for receiving relevant data $\mathcal{V}_{\alpha_i}$ from another agent $\alpha_i \in \mathcal{E}_{\text{COLL}}$ executing the same Collective Capability. RECEIVE($c^v_{\text{COLL}}, \mathcal{V}_{\alpha_i}$) updates the values for these other agents stored in a shared map $\text{M}^{\mathcal{E}_{\text{COLL}}} := \langle \alpha \in \mathcal{E}_{\text{COLL}}, M^\alpha \rangle$ holding the most recent values M^α received from all agents $\alpha \in \mathcal{E}_{\text{COLL}}$. To enable the exchange of data locally for each agent in the collective, the active and passive part of each c^v_{COLL} share this map. This means, when receiving $\mathcal{V}_{\alpha_{i\neq j}}$ in $c^{v:\text{PAS}}_{\text{COLL}}$, an agent α_j can update the entries referenced in $\mathcal{V}_{\alpha_i}$ concerning α_i in $\text{M}^{\mathcal{E}_{\text{COLL}}}$ and subsequently has access to the data in $c^{v:\text{ACT}}_{\text{COLL}}$. In our code snippets, we indicate that α_j executing c^v_{COLL} sends $\mathcal{V}_{\alpha_j}$ to a specific other agent α_i executing the same instance of c^v_{COLL} with α_i.SEND($c^v_{\text{COLL}}, \mathcal{V}_{\alpha_j}$).

7.5.2 Coordinating and Executing Collective Capabilities

Like for other capabilities, we can define different termination types for Collective Capabilities. They can terminate their execution internally or require external termination. We can define

Algorithm 6 $c^{v:\text{PAS}}_{\text{COLL}}(\text{M}^{\mathcal{E}_{\text{COLL}}})$

1: $\text{M}^{\mathcal{E}_{\text{COLL}}}[\alpha_i] \leftarrow \text{RECEIVE}(c^v_{\text{COLL}}, \mathcal{V}_{\alpha_i})$ *# updates the map with values sent by* α_i

Algorithm 7 $c^{v:\text{FIN-COORD}}_{\text{COLL}}(\text{F:AGGR}_{c^v_{\text{COLL}}}, \text{F:TERM}_{c^v_{\text{COLL}}})$

1: $\text{R}_{\text{AGGR}} \leftarrow \text{F:AGGR}_{c^v_{\text{COLL}}}(\text{M}^{\mathcal{E}_{\text{COLL}}})$ *# aggregates the ensemble's current measurements*
2: $\text{TERM} \leftarrow \text{F:TERM}_{c^v_{\text{COLL}}}(\text{R}_{\text{AGGR}})$ *# decide for termination using the aggregated result*
3: **if** TERM **then**
4: $\text{STORE}(\text{R}_{\text{AGGR}})$ *# if terminating, store the result for external evaluation*
5: **for** $\alpha_i \in \mathcal{E}_{\text{COLL}}$ **do**
6: $\alpha_i.\text{SEND}(c^v_{\text{COLL}}, \text{TERM})$ *# broadcast the termination decision in the ensemble*

Algorithm 8 $c^{v:\text{FIN-PART}}_{\text{COLL}}(\text{TERM}^{\mathcal{E}_{\text{COLL}}})$

1: $\text{TERM}^{\mathcal{E}_{\text{COLL}}} \leftarrow \text{RECEIVE}(c^v_{\text{COLL}}, \text{TERM})$

Figure 7.4: Program snippets agents need to implement for being able to participate in Collective Capability.

termination criteria with appropriate parameters for some swarm behavior, e.g., executing a Collective Capability implementing a PSO can terminate itself when all agents in the collective gather within a certain distance [Zhang et al., 2015]. For other collective behavior, e.g., achieving the equal distribution of robots in a given area with the triangle algorithm [Li et al., 2009], we do not want to define such criteria (e.g., for achieving the continuous surveillance of that area) or even cannot do it at all (e.g., for steering a swarm in one direction with guided flocking [Çelikkanat et al., 2009]) and thus rely on an external signal for termination. Besides defining when to terminate a Collective Capability c^v_{COLL}, we also require to quantify the emergent effect of executing c^v_{COLL} and store it for up-following evaluation like we do with the results originating from other capability executions. For PSO, e.g., we finally want to determine the position the highest concentration of a parameter an ensemble was searching for was measured. In this case, we can calculate the position of relevance as the ensemble's centroid when the geometrical diameter of the swarm, i.e., the euclidean distance between the $\alpha_i, \alpha_j \in \mathcal{E}_{\text{COLL}}$ having the greatest distance between each other, gets lower than a user-defined threshold. For such calculations and to determine termination for Collective Capabilities therewith, we give the agent acting as PFC for Ensemble Programs another program $c^{v:\text{FIN-COORD}}_{\text{COLL}}$ (cf. Algorithm 7) at hand that can aggregate the respective information for collectives.

Concerning the results of Non-Collective Capabilities, the PFC program from Section 7.4 acts as a pass-through station for results originating from any instruction executed by the agents in the ensemble. If advised, the PFC stores individual results in a WS and evaluates data stored there when necessary, e.g., for deciding on the current program's progress (cf. Section 7.4.2.2). To determine the termination of a Collective Capability c^v_{COLL}, we now enable the PFC also to aggregate, analyze, and post-process the intermediate results from Collective

Algorithm 9 $c^{v:\text{ACT}}_{\text{PROTEASE}}$ $(C_{\text{PROTEASE}}, \text{CALC}_{\text{PROTEASE}}, \mathcal{E}_{\text{PROTEASE}})$

```
1: repeat
2:   for each c_i ∈ C_PROTEASE parallel do
3:     M^SELF[c_i] ← EXEC(c_i)      # execute all relevant capabilities and store the results
4:   M^{E_COLL}[SELF] ← M^SELF      # store local results in the map for all ensemble results
5:   for each α_i ∈ E_PROTEASE parallel do
6:     α_i.SEND(c^v_PROTEASE, M^SELF)      # distribute stored results in the ensemble
7:   PAR_{c^p_MV-VEL} ← CALC_PROTEASE(M^{E_COLL})      # calculate the new movement vector
8:   EXEC(c^p_MV-VEL(PAR_{c^p_MV-VEL}))      # update the current movement vector
9: until TERM      # decide on termination using the received value
```

Figure 7.5: The Collective Capability c^v_{PROTEASE} as instance of a virtual capability.

Capabilities before eventually storing their result into WS with an additional procedure. Because we ensure that the agent realizing PFC for a Collective Capability is also a part of $\mathcal{E}_{\text{COLL}}$, i.e., the agent also executes the respective $c^{v:\text{ACT}}$, it can receive values sent by other agents $\alpha \in \mathcal{E}_{\text{COLL}}$ and thus has access to $\text{M}^{\mathcal{E}_{\text{COLL}}}$. By using an aggregation function $\text{F:AGGR}_{c^v_{\text{COLL}}}$ taking $\text{M}^{\mathcal{E}_{\text{COLL}}}$ as input parameter that is specific for each c^v_{COLL}, we can quantify the emergent effect every time the entries in $\text{M}^{\mathcal{E}_{\text{COLL}}}$ change (Line 1 in Algorithm 7). If the termination criteria ($\text{F:TERM}_{c^v_{\text{COLL}}}$ in Algorithm 7) holds for the current result (Line 2 in Algorithm 7), the PFC can store that result in the distributed storage (Line 4 in Algorithm 7) and distribute the current termination state TERM within the collective $\mathcal{E}_{\text{COLL}}$ (Line 6 in Algorithm 7).

To enable agents participating in the collective executing the respective $\mathcal{E}_{\text{COLL}}$ to receive that termination state TERM, we also extend the EPU programs with extra functionality. Each agent implementing an EPU program thus also offers a respective service $c^{v:\text{FIN-PART}}_{\text{COLL}}$ (cf. Algorithm 8) to receive the PFC program's termination signal TERM with $\text{RECEIVE}(c^v_{\text{COLL}}, \text{TERM})$. The service $c^{v:\text{FIN-PART}}_{\text{COLL}}$ shares TERM with the active part $c^{v:\text{ACT}}_{\text{COLL}}$ of c^v_{COLL} in $\text{TERM}^{\mathcal{E}_{\text{COLL}}}$, which we use to stop the execution of c^v_{COLL}. We can determine and distribute information on termination of the respective Collective Capability and receive external input from the user if the Collective Capability requires such.

7.5.3 Defining PROTEASE(ATGC) as a Collective Capability

We now demonstrate how we can integrate the Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE) (cf. Section 7.3) as such Collective Capability. Therefore, we provide an algorithmic description of the active part $c^{v:\text{ACT}}_{\text{PROTEASE}}$ of the respective Collective Capability c^v_{PROTEASE} in Algorithm 9.

Like we propose for PROTEASE in Section 7.3, we require different input defined as A (the aggregation function), T (the termination function), G (the group of agents forming the collective), and C (the calculation function) for correctly executing c^v_{PROTEASE}. The parameters A and T are relevant for the PFC program's execution in the general procedure $c^{v:\text{FIN-COORD}}_{\text{COLL}}$ we use for each Collective Capability (cf. Algorithm 7), i.e., A is defined by $\text{F:AGGR}_{c^v_{\text{COLL}}}$ and T is defined by $\text{F:TERM}_{c^v_{\text{COLL}}}$. The parameters G and C are relevant for the EPU program's execution in the specific $c^{v:\text{ACT}}_{\text{PROTEASE}}$ $(C_{\text{PROTEASE}}, \text{CALC}_{\text{PROTEASE}}, \mathcal{E}_{\text{PROTEASE}})$, i.e., G is defined by $\mathcal{E}_{\text{PROTEASE}}$.

C is defined by $\text{CALC}_{\text{PROTEASE}}$ combined with the explicitly named capabilities C_{PROTEASE} and the implicitly named capability $c^{p}_{\text{MV-VEL}}$ for correctly executing $\text{CALC}_{\text{PROTEASE}}$. This comes as c^{v}_{PROTEASE} is a virtual capability combined from $c^{p}_{\text{MV-VEL}}$ and the respective capabilities named in $\text{CALC}_{\text{PROTEASE}}$).

In a first step, each agent executing $c^{v:\text{ACT}}_{\text{PROTEASE}}$ measures and remembers relevant values according to the set of capabilities C_{PROTEASE} included in the parameters of $c^{v:\text{ACT}}_{\text{PROTEASE}}$ in parallel (cf. Line 3 in Algorithm 9). After finishing the execution of all capabilities in case of self-terminating capabilities or after starting to execute non-self-terminating capabilities respectively (cf. their difference in Section 7.4.1), agents executing $c^{v:\text{ACT}}_{\text{PROTEASE}}$ in parallel exchange their so-generated local results M^{SELF} with all other agents in the current collective $\mathcal{E}_{\text{PROTEASE}}$ that execute the same instance of c^{v}_{PROTEASE} (cf. Line 6 in Algorithm 9). Each agent $\alpha \in \mathcal{E}_{\text{PROTEASE}}$ remembers the results transmitted by other agents in the Collective Capability's locally shared map $\text{M}^{\mathcal{E}_{\text{COLL}}}$ that holds the most recent values for all neighbors including itself (cf. Line 4 in Algorithm 9). By using this aggregated measurements $\text{M}^{\mathcal{E}_{\text{COLL}}}$, each agent then is able to determine the necessary adaption to its current trajectory (cf. Line 8 in Algorithm 9) for achieving the intended specific swarm behavior encapsulated in the calculation function from the parameters A, T, G, C (cf. Line 7 in Algorithm 9). As all agents in $\mathcal{E}_{\text{PROTEASE}}$ repeatedly execute this behavior until a specific termination criteria TERM holds (passed over to $c^{v:\text{FIN-PART}}_{\text{COLL}}$ from the coordinator, cf. Algorithms 7 and 8), they achieve the specific swarm algorithm's emergent effect collectively (cf. Line 9 in Algorithm 9). Thus, by adjusting $\text{CALC}_{\text{PROTEASE}}$ in particular, we can produce different swarm behavior that would require an individual implementation otherwise. Using the combination of $c^{v:\text{ACT}}_{\text{PROTEASE}}$ on the one hand and $c^{v:\text{FIN-COORD}}_{\text{COLL}}$ on the other, we can let the executing collective determine that a result is available and the respective value of that result, making trajectory-based swarm behavior usable in a goal-oriented fashion in Ensemble Programs.

7.5.4 An Interface for External Collective Programming Languages

Besides the Collective Capability c^{v}_{PROTEASE} we describe in Section 7.5.3, we also provide a possibility for integrating collective behavior defined with other approaches for programming aggregates/swarms/ensembles like Protelis [Pianini et al., 2015] or Buzz [Pinciroli and Beltrame, 2016] into our mechanism for task orchestration. Therefore, we offer a Collective Capability providing an interface for integrating external programs to the system's user. This interface can take programs encapsulating collective behavior defined with other programming languages, enriches them with some additional information required by our mechanism, and then executes those programs as Collective Capabilities.

We do this by introducing a Collective Capability for every external collective programming approach we want to include in our approach MAPLE. Here, we describe the scheme such Collective Capabilities need to follow during this integration. In contrast to c^{v}_{PROTEASE}, where we need to define the actual calculation $\text{CALC}_{\text{PROTEASE}}$ within the host system (e.g., the reference implementation of Multipotent System we provide in Section 3.3.2) and its respective programming language we are not restricted to that when using a specific c^{v}_{EXT}. Instead, we encapsulate necessary information in the external program itself written directly in the respective external programming language. To enable such, we need to define the interface for the communication of that programming language's execution environment and the host system's implementation in an abstract Collective Capability c^{v}_{EXT}. External programs then self-define how values we generate within the host system are used and transformed back into

Algorithm 10 $c_{\text{EXT}}^{v:\text{ACT}}$ $(\text{PROG}_{\text{EXT}}, \text{PC}_{\text{EXT}}, \mathcal{E}_{\text{EXT}})$

1: **repeat**
2: $\text{M}_{\text{SNAP}}^{\mathcal{E}_{\text{COLL}}} \leftarrow \text{M}^{\mathcal{E}_{\text{COLL}}}$ *# create a snapshot of the current ensemble values*
3: $\langle C_{\text{EXT}}, \text{TERM}_{\text{EXT}}, \text{PC}_{\text{EXT}}, \mathcal{V}_{\text{EXT}} \rangle \leftarrow \text{PROG}(\text{PC}_{\text{EXT}}, \text{M}_{\text{SNAP}}^{\mathcal{E}_{\text{COLL}}})$ *#execute the program*
4: **for each** $c_i \in C_{\text{EXT}}$ **parallel do**
5: $\text{M}^{\text{SELF}}[c_i] \leftarrow \text{EXEC}(c_i)$ *#execute capabilities required by the program*
6: $\text{M}^{\mathcal{E}_{\text{COLL}}}[\text{SELF}] \leftarrow \text{M}^{\text{SELF}}$ *#store results for next iteration of the program*
7: **for each** $\alpha_i \in \mathcal{E}_{\text{EXT}}$ **parallel do**
8: $\alpha_i.\text{SEND}(c_{\text{EXT}}^{v}, \mathcal{V}_{\text{EXT}})$ *#distribute relevant data of the program*
9: **until** $\text{TERM}_{\text{EXT}} \vee \text{TERM}$ *#check termination set by the program or coordinator*

Figure 7.6: Realizing an adapter for external collective programming approaches with c_{EXT}^{v} as instance of a virtual capability.

new instructions for the host system using the external programming approach. Like for any Collective Capability, we enable each instance of c_{EXT}^{v} to execute other already existing capabilities $c \in \mathcal{C}$ of the host system, i.e., choose respective parameters and read results from those capabilities' execution that we store in $M^{\mathcal{E}_{\text{COLL}}}$ through the defined interface (Line 5 in Algorithm 10). This way, a user can program new complex behavior PROG_{EXT} in the external programming language while also using already available functionality provided by capabilities $c \in \mathcal{C}$ within the host system. The programmer only needs to know the interface to relevant $c \in \mathcal{C}$ and does not require further knowledge of the underlying host system and the individual implementation of capabilities. For its execution, the respective c_{EXT}^{v} then uses PROG_{EXT} as an additional parameter (cf. Algorithm 10). This way, and to change the behavior of c_{EXT}^{v}, the programmer can dynamically exchange the external program at run-time.

With the start of the capability execution within the active part of each $c_{\text{EXT}}^{v:\text{ACT}}$, we run PROG_{EXT} from its entry point by handing over a program pointer PC_{EXT} and a snapshot of the current state of $\text{M}^{\mathcal{E}_{\text{COLL}}}$ (initially empty, Line 2 and 3 in Algorithm 10). When the execution of PROG_{EXT} for that initial program counter stops, we require it to return a data vector $\langle C_{\text{EXT}}, \text{TERM}_{\text{EXT}}, \text{PC}_{\text{EXT}}, \mathcal{V}_{\text{EXT}} \rangle$ encapsulating instructions from the external program to the host system. We describe the entries of the data vector in the following.

- The first entry of this data vector indicates whether the external program's control flow requires that capabilities defined in C_{EXT} get executed in the following by the host system (Line 4 and 5 in Algorithm 10).
- The second entry of this data vector $\langle C_{\text{EXT}}$ determines, whether PROG_{EXT} already reached its termination criteria TERM_{EXT} and the execution of c_{EXT}^{v} can be finished (Line 9 in Algorithm 10).
- The third entry of this data vector determines, what the next program counter PC_{EXT} is if TERM_{EXT} does not hold, i.e., the external program did not already finish its execution and thus requires to continue running.
- Because information on which values need to be exchanged within the ensemble $\mathcal{E}_{\text{EXT}}$ is encapsulated in PROG_{EXT} but the distribution itself is performed by the host system's

communication interface, the data vector determines values to distribute that way in a fourth entry $\mathcal{V}_{\text{EXT}}$ (Line 7 and 8 in Algorithm 10).

While TERM_{EXT} does not hold and no termination signal is received from the PFC program in $c^{v:\text{FIN-PART}}$ (cf. Section 7.5.2), the execution of c^{v}_{EXT} continues to execute PROG_{EXT} with the respectively updated PC_{EXT} in the following iteration. Thereby, it uses an updated version of $\text{M}^{\mathcal{E}_{\text{COLL}}}$ (Line 2 in Algorithm 10) containing the results of the latest local executions of capabilities (Line 6 in Algorithm 10) as well as such results received from other agents in $\alpha \in \mathcal{E}_{\text{EXT}}$ in $c^{v:\text{PAS}}_{\text{EXT}}$ meanwhile. Each PROG_{EXT} adhering to this convention thus can access the set of locally available capabilities and use the communication interface of the host system to exchange data within the respective collective.

7.6 Evaluation

We perform our evaluations for demonstrating a working program execution and the integration of Collective Capabilities into Ensemble Programs in three stages. First, we evaluate the Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE) by investigating its generality of integrating different swarm behavior we can find in current literature. Second, we demonstrate how our approach of encapsulating PROTEASE into a Collective Capability enables its goal-oriented usage and how we can use it seamlessly integrated into the execution of Ensemble Programs. We perform this evaluation using our reference implementation of Multipotent System we introduced in Section 3.3. Third, we demonstrate how we can integrate other approaches for programming collective behavior into Ensemble Programs by encapsulating them as instances of the abstract Collective Capability c^{v}_{EXT}. We do this with the example of Protelis [Pianini et al., 2015].

7.6.1 Evaluating the Generality of PROTEASE

We validate the generality of the Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE) using different parameters A, T, G, C for realizing different emergent effects generated by a respectively changed swarm behavior. We therefore use the *NetLogo multi-agent programmable modeling environment*[3] whose execution model we first briefly describe in Section 7.6.1.1 before we illustrate our theoretical and experimental results.

In the first set of examples, we demonstrate how to generate beneficial swarm behavior and how toapply it for using it in our case study on *Dealing with Gas Accidents* (cf. Section 2.5). In a Major Catastrophe Handling scenario, rescue forces might want to I. gather all agents in the system, II. move them collectively to the area where a gas accident happened, III. search for the source of the gas leak, and IV. survey the area close to the leak. We can produce each desired behavior in the described steps I - IV using PROTEASE executed with a different set of parameters we describe in the following Section 7.6.1.2.

The second set of examples illustrates further beneficial applications of PROTEASE we can apply in other case studies in Section 7.6.1.3. For all executions of PROTEASE producing the desired emergent effect for achieving, we assume the following: For each result of $\text{CALC}_{\text{PROTEASE}}$, we normalize (NORM()) the resulting distance (DIST()) vector originating from the agents current position POS_{α} and scale it with the agents maximum velocity with ν. We further assume a working collision avoidance system provided by the respective agents to neglect

[3] NetLogo download on `https://ccl.northwestern.edu/netlogo/download.shtml`

collisions in our simulation environment. Further, when introducing the respective parameters for different swarm behavior, we already use the terminology we introduced for the Collective Capability c^{v}_{PROTEASE} in Section 7.6.2 to avoid confusion.

We illustrate the achieved results with respective figures here and video materials online. The video materials we generated using the NetLogo simulation and 3D-visualization can be found on GitHub and YouTube[4].

7.6.1.1 The Netlogo Testing Environment

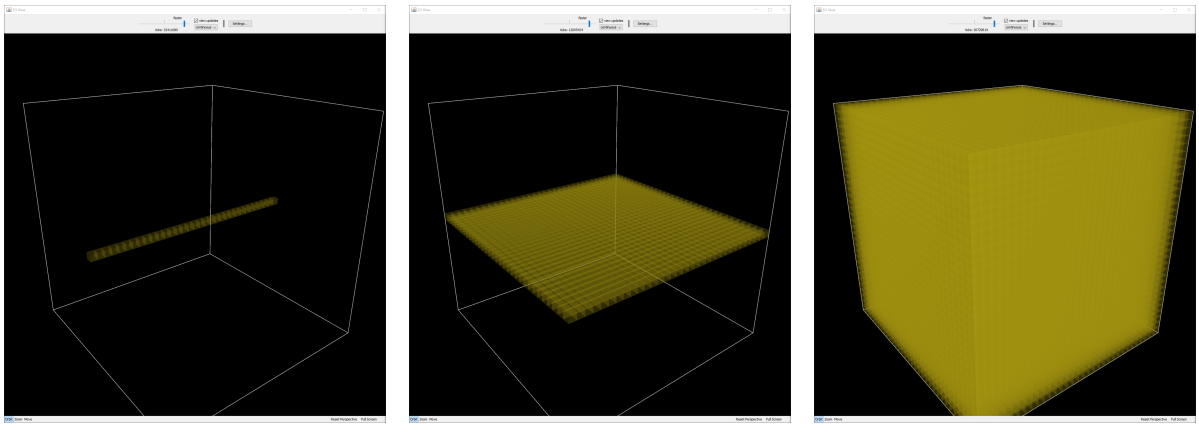
NetLogo [CCL, 2020] is a programming environment designed for easy setup multi-agent system simulations. Compared to a real-world setting, it abstracts from many details, reduces complexity by appropriate discretization, and lowers the hurdle for prototyping collective and swarm behavior. It offers possibilities to simulate dynamic and static elements, which we use to simulate *agents* and the *environment*. Moreover, it supports a logical *time* model.

Time Progress in time is designed discretely in NetLogo. For every individual simulation run, a tick counter is initialized with zero. When executing the simulation, the tick counter is increased, and thus, time progresses logically. In every tick, the environment and the simulation can change their internal states by executing their implemented functionality once.

Environment In general, NetLogo supports 2-dimensional and 3-dimensional environments. To demonstrate the feasibility of PROTEASE, we make use of the Netlogo distribution supporting 3-dimensional environments. To reduce complexity concerning the possible state space of the environment, NetLogo separates the static environment discretely in cubes (called *patches* internally). The environment thus consists of vertically stacked geometric planes (z-axis) consisting of cubes strung together horizontally (x-axis and y-axis) each (cf. Figure 7.7). Single cubes can store and exchange information, e.g., the concentration and dissemination of a specific parameter like that of a particular gas over time. The system designer can define the range in which one cube from the environment can propagate information to other cubes. Thus, not only directly adjacent cubes can exchange information but also cubes that are further away from each other. We can use this to simulate a faster spread of information in the environment when necessary, e.g., to simulate a different relative speed of different parameters.

Agents Agents (called *breed* internally) are located and oriented within this environment. They are aware of environmental information and can use it for internal processing. Because agents are simulated as dynamic elements, they can move in the environment and traverse different cubes. The speed agents can move with can be defined by the system's designer. Further, agents can request values stored in the environment, e.g., read the current concentration of parameters, and thus implicitly have sensors for any possible value. Agents can also communicate with each other and exchange any information they are aware of. The system's designer can set communication range and sensing range, e.g., to simulate more close to realistic sensors and actuators. In our figures, agents are represented by orange arrows indicating their position and orientation in the 3-dimensional environment.

[4] `https://github.com/isse-augsburg/ensemble-programming` or `https://github.com/kosakoliver/ensemble-programming` or `https://www.youtube.com/user/ISSELabs`



(a) row of cubes on x-axis (b) plane of cubes on x-/y-axis (c) stacked planes on z-axis

Figure 7.7: 3-dimensional NetLogo environment consisting of cubes strung together in rows horizontally (x-axis and y-axis) and in planes vertically (z-axis). Cubes are marked with shaded yellow color, separated by a yellow grid.

General Parameters For Choosing Swarm Behavior and Initializing / Modifying the System The NetLogo environment we designed to demonstrate the generality of PROTEASE consists of general parameters (upper part in Figure 7.8) for setting up the simulation environment, i.e., *setup*, *go*, *spawn-new-agent*, and *remove-one-agent* buttons, and modifying agent-internal properties, i.e., define their maximum velocity *max-velocity* and communication radius *neighborhood-radius* and define their initial number *agent-count*. It further provides the possibility for user input during execution to change the parameters of PROTEASE, i.e., switches between different swarm behaviors.

Adjustable Run-Time Parameters for the Different Swarm Behaviors In the lower part of Figure 7.8, we provided some additional parameters to the user for influencing the environment and some constants necessary for specific swarm behavior allowing for their rapid prototyping. For the triangle swarm behavior, e.g., this contains the constants *side-length* for defining the desired distance between agents executing the individual swarm behavior and *form-zcor* to modify the height where the swarm behavior should establish externally. Other parameters can modify the environment values an agent measures while executing a specific swarm behavior, e.g., the parameters defining the form for the fill-form and shape-form swarm behaviors we describe in the following.

7.6.1.2 Application of PROTEASE in Major Catastrophe Scenarios

We now describe the application of PROTEASE in the use case of *Dealing with Gas Accidents* using different parameters for producing different swarm behavior that we can beneficially apply for in the scenario. The foremost goal we want to achieve in the experiments in this section is to demonstrate the generality of PROTEASE concerning its ability for producing different swarm behaviors by only changing its parameters (videos on GitHub and YouTube). Nevertheless, we also describe how we can define parameters $\text{TERM}_{\text{PROTEASE}}$ and $\text{AGG}_{\text{PROTEASE}}$ for each behavior we produce with PROTEASE. Thus, we focus on changing $\text{CALC}_{\text{PROTEASE}}$ and

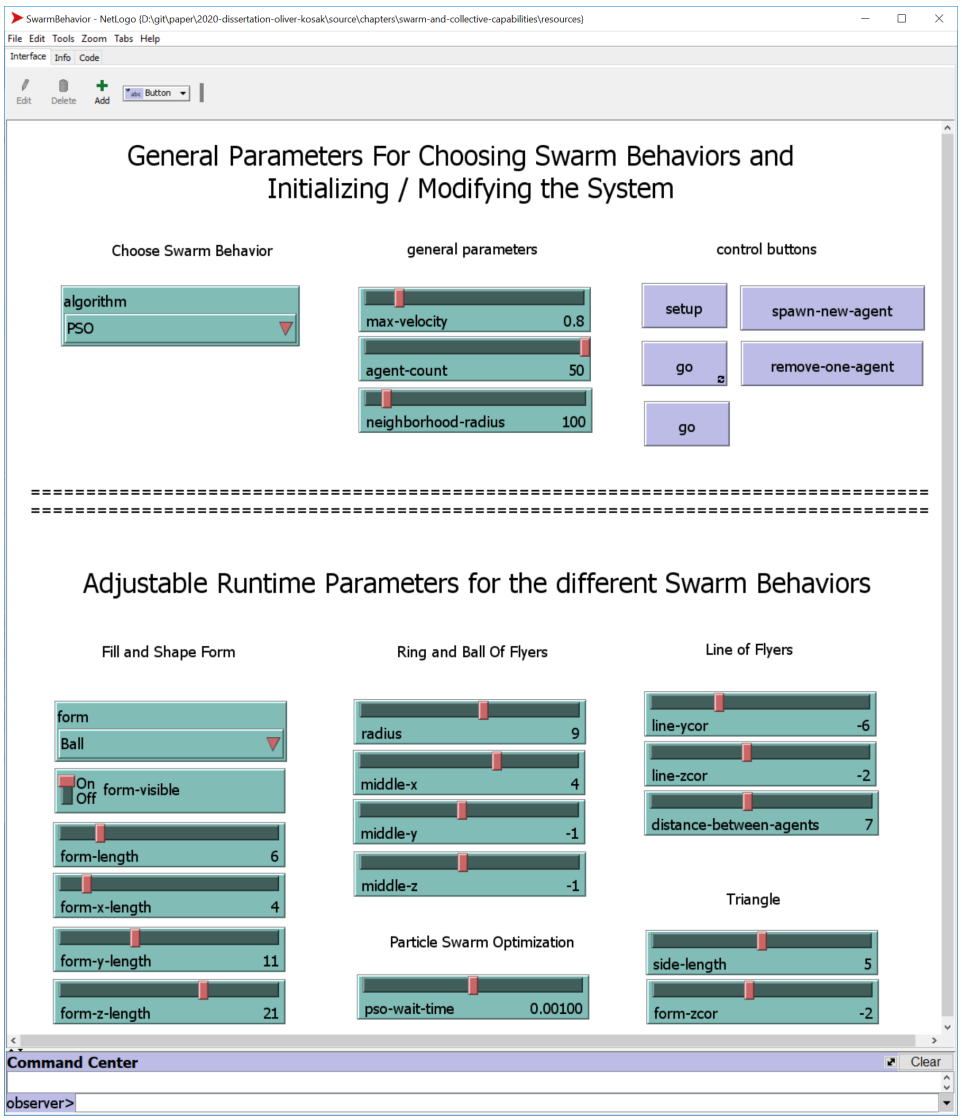
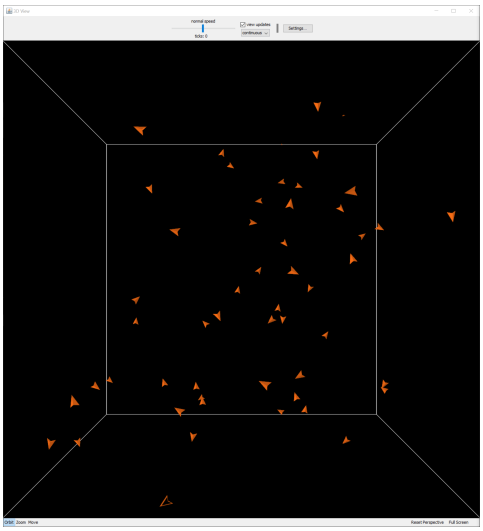


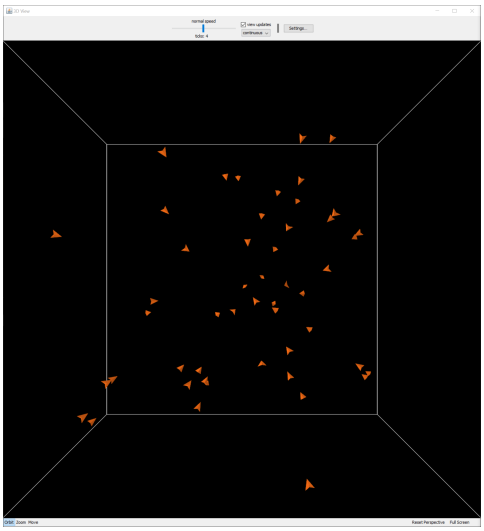
Figure 7.8: The Netlogo simulation environment to demonstrate the ability to achieve different swarm behavior with swarm capability by executing it with different sets of parameters.

$\text{GROUP}_{\text{PROTEASE}}$ here and investigating the effect of $\text{TERM}_{\text{PROTEASE}}$ and $\text{AGG}_{\text{PROTEASE}}$ in more detail in the next section we use to evaluate the goal-oriented integration of PROTEASE in task orchestration.

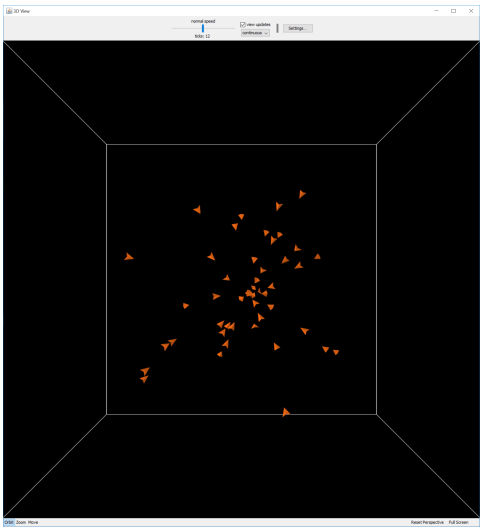
I) Call Collectives Together with the Gathering Swarm Behavior Suppose the scenario that all agents initially are distributed randomly in the environment (cf. Figure 7.9a). If we want to gather them at the desired position before collectively moving them to a goal destination, we can execute PROTEASE using appropriate parameters. In our experiment (video *PROTEASE-NetLogo-Gathering* on GitHub and YouTube), we define parameter $\text{GROUP}_{\text{PROTEASE}}$ to cover all other agents in the communication range *neighborhood-radius* that are also simulated in the NetLogo environment, ensuring that every agent communicates with at least one



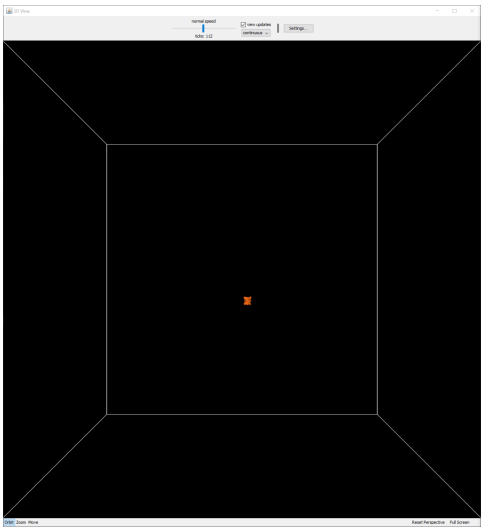
(a) randomized setup for gathering behavior



(b) orientating to the collective's centroid



(c) moving to the collective's centroid



(d) gathering at the collective's centroid

Figure 7.9: Image exports from of the NetLogo simulation environment while using PROTEASE for achieving a *gathering swarm behavior* in four states from a top down perspective.

other agent every time. For executing the calculation function $\text{CALC}_{\text{PROTEASE}}$ we use for determining an agent's new trajectory in PROTEASE, we require each agent to be able to measure its position, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{POS}}\}$ in the notation we use in Algorithm 9. Because each agent receives position measurements of all other agents defined in $\text{GROUP}_{\text{PROTEASE}}$ it can calculate the desired moving vector using the centroid CENTR() defined by all agents in $\text{GROUP}_{\text{PROTEASE}}$ it receives measurements from, i.e.,

$$\text{CALC}_{\text{PROTEASE}} := \nu \cdot \text{NORM}(\text{DIST}(\text{POS}_\alpha, \text{CENTR}(\text{M}^{\mathcal{E}}_{\text{PROTEASE}}[*][c^p_{\text{POS}}])))$$

the notation we use in Algorithm 9. We see how this modifies the trajectories of all agents in Figure 7.9b, at first modifying their heading directories towards the centroid. In Figure 7.9c

we then can also see the progress made by PROTEASE, moving all agents towards the centroid. The execution of PROTEASE used for a gathering then can be terminated when all positions of agents defined in $\text{GROUP}_{\text{PROTEASE}}$ are close enough to each other. Thus each agent can terminate the execution when the diameter DIAM() of G is below a user-defined threshold x. Using the terminology of Algorithms 7 and 9, the measurements available in $\text{M}^{\mathcal{E}_{\text{COLL}}}$ can be used therefore, i.e.,

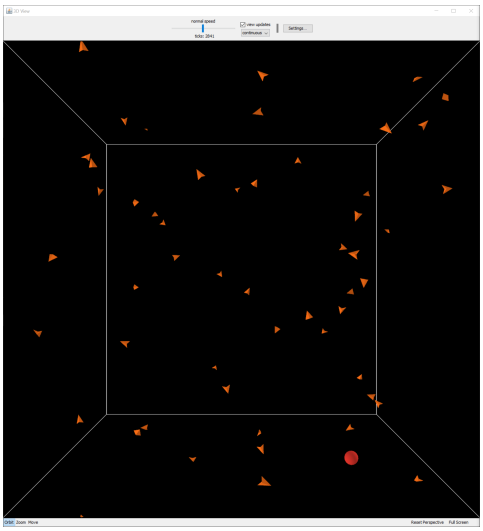
$$\text{TERM}_{\text{PROTEASE}} := \text{DIAM}(\text{M}^{\mathcal{E}}_{\text{PROTEASE}}[*][c^p_{\text{POS}}]) \leq x$$

where $*$ stands for all entries concerning the current positions of the agents in $\text{GROUP}_{\text{PROTEASE}}$ whose position measurements are included in $\text{M}^{\mathcal{E}_{\text{PROTEASE}}}$. We can see that state in Figure 7.9d, where all agents reached the Rendez-Vouz position, i.e., the dynamically calculated centroid. Both, DIAM() and CENTR() only require information concerning the position of agents in $\mathcal{E}_{\text{PROTEASE}}$, thus results from executing c^p_{POS} stored in $\text{M}^{\mathcal{E}_{\text{COLL}}}$ are sufficient therefore. The result of PROTEASE is the respective gathering position and can be derived using a respective aggregation function $\text{AGG}_{\text{PROTEASE}}$ again using the data provided in $\text{M}^{\mathcal{E}_{\text{PROTEASE}}}$.

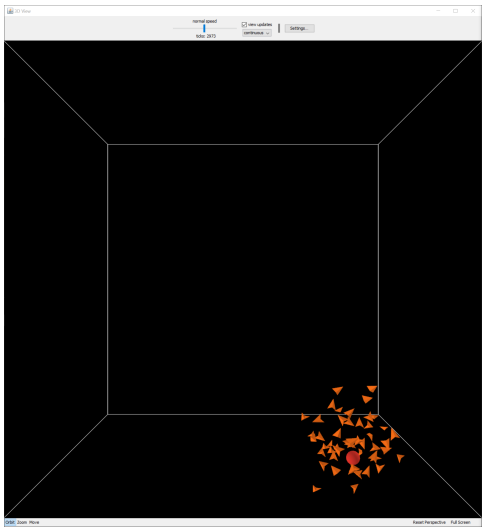
II) Moving Collectives with the Guided Boiding Swarm Behavior After we have gathered all agents at one position, we now might want to move the whole collective at once to another location relevant to the scenario. We, therefore again, can use PROTEASE with modified parameters to produce an emergent effect close to that described as "boiding" by Reynolds [1987] (video *PROTEASE-NetLogo-Guided-Boiding* on GitHub and YouTube). By slightly adapting this flocking behavior by introducing one specific entity in the collective that we can control externally, we can produce the desired effect (cf. Figure 7.10). Again, we execute PROTEASE with the parameter $\text{GROUP}_{\text{PROTEASE}}$ set to cover all other agents within the communication range. Initially, all agents described by $\text{GROUP}_{\text{PROTEASE}}$ are randomly distributed in the environment (cf. Figure 7.10a). Compared to gathering, we now extend the set of measurements we use in $\text{CALC}_{\text{PROTEASE}}$ when calculating each agent's new trajectory with additional velocity measurements, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{M-POS}}, c^p_{\text{MV-VEL}}\}$. The calculation function $\text{CALC}_{\text{PROTEASE}}$ then uses local measurements and those of other agents transmitted to the agent applying them to the well known "urges" concept of Reynolds [1987]: We appropriately weight the three urges for the cohesion COH() of agents, the separation SEP() from the closest neighbor, and the alignment ALI() of all agents' trajectory headings defined by $\text{GROUP}_{\text{PROTEASE}}$:

$$\begin{aligned}\text{CALC}_{\text{PROTEASE}} :=& \omega_1 \cdot \text{SEP}(\text{M}^{\mathcal{E}}_{\text{PROTEASE}}[*][c^p_{\text{M-POS}}]) + \\ & \omega_2 \cdot \text{COH}(\text{M}^{\mathcal{E}}_{\text{PROTEASE}}[*][c^p_{\text{M-VEL}}]) + \\ & \omega_3 \cdot \text{ALI}(\text{M}^{\mathcal{E}}_{\text{PROTEASE}}[*][c^p_{\text{MV-VEL}}])\end{aligned}$$

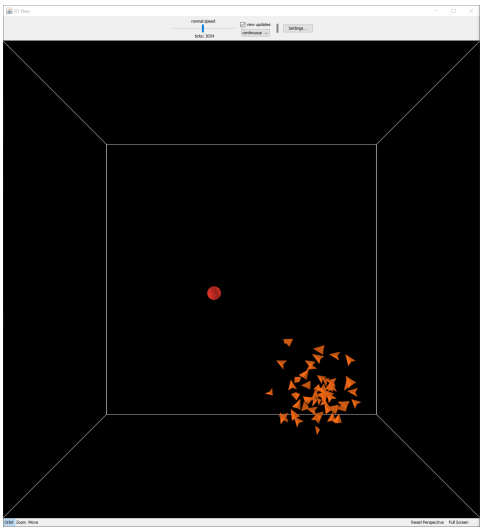
To guide the agents in $\text{GROUP}_{\text{PROTEASE}}$ to the goal location we exploit how agents evaluate $\text{M}^{\mathcal{E}_{\text{PROTEASE}}}$ for adapting their trajectory in $\text{CALC}_{\text{PROTEASE}}$. We add an entry for a non-ensemble member in that measurements, i.e., extend $\text{GROUP}_{\text{PROTEASE}}$ with a dedicated leader agent controlled by the user (red ball in Figure 7.10). This agent also sends updates of its position measurements to all other agents covered by $\text{GROUP}_{\text{PROTEASE}}$. Because all agents in $\text{GROUP}_{\text{PROTEASE}}$ use the complete map $\text{M}^{\mathcal{E}_{\text{PROTEASE}}}$, the emergent effect is what we aim for. First, all agents thus form a flock near the user-controlled leader (cf. Figure 7.10b). We then can move the leader towards the goal location, triggering all agents in $\text{GROUP}_{\text{PROTEASE}}$ to start moving towards that location because of the collective's cohesion urge (cf. Figure 7.10c). When all agents in



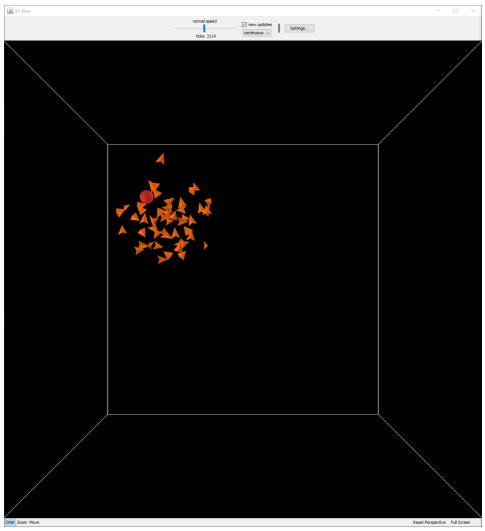
(a) randomized setup for guided boiding



(b) boiding around the leader agent



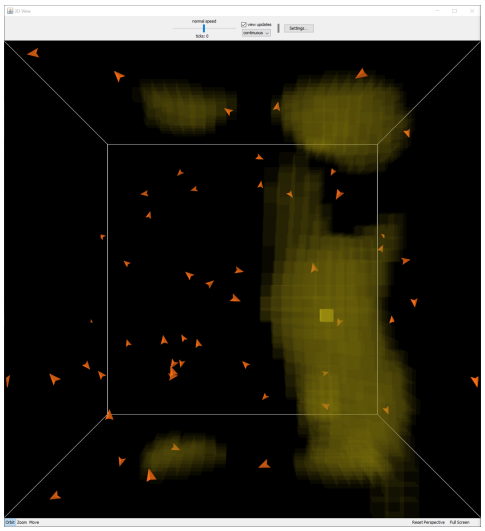
(c) moving the leader agent



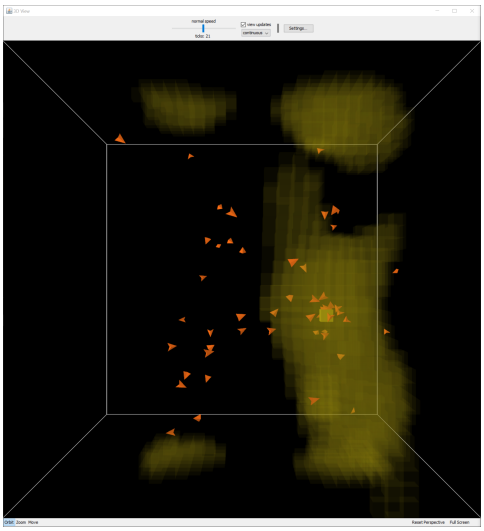
(d) reforming at the leader's new position

Figure 7.10: Image exports from the NetLogo simulation environment while executing PROTEASE to achieve a *guided boiding swarm behavior* in four states from a top down perspective. The guiding user-controlled agent is represented by a red ball.

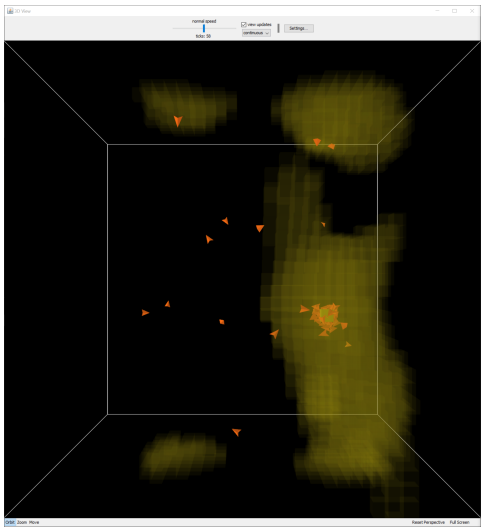
$\text{GROUP}_{\text{PROTEASE}}$ have finally reached the goal location, they again start to form a flock near the leader (cf. Figure 7.10d). Because all non-leader agents participating in the swarm behavior have no information when they have reached this goal location, we cannot define any termination function $\text{TERM}_{\text{PROTEASE}}$ using only swarm-internal information. Nevertheless, as we already involve the user in the execution of PROTEASE (i.e., by steering the leader), we can instead easily rely on a user-generated signal for terminating the execution of PROTEASE. Again, we can define the result of the described execution of PROTEASE, i.e., the aggregation function $\text{AGG}_{\text{PROTEASE}}$, as the centroid calculated on the last position measurement by agents covered with $\text{GROUP}_{\text{PROTEASE}}$ (in a final state $\text{GROUP}_{\text{PROTEASE}}$ covers all agents in the swarm) if



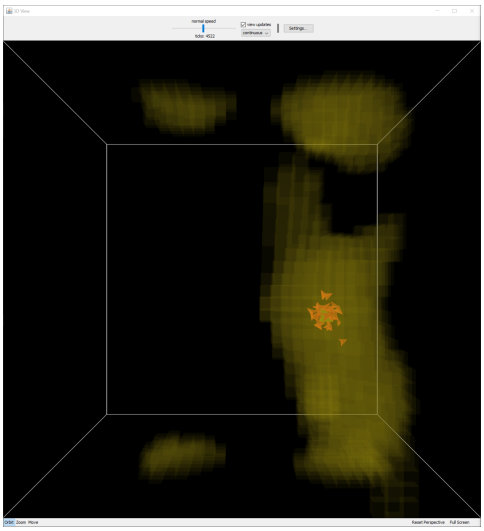
(a) randomized setup for particle swarm optimization



(b) first agents detect the POI and move towards it



(c) neighbors get aware of the POI and also move there



(d) all agents aggregate around the POI

Figure 7.11: Image exports from of the NetLogo simulation environment during executing PROTEASE for achieving a *PSO swarm behavior* in four states from a top down perspective. The concentration of the parameter of interest is indicated by yellow colored environment (higher color intensity indicates higher parameter concentration).

we require it.

III) Searching a Point of Interest (POI) with the Adapted Particle Swarm Optimization Behavior Suppose all agents now have reached the desired goal location. We then want them to execute another swarm behavior for searching for the highest concentration of a specific parameter of interest. We indicate the concentration of such generic parameter

with different intensities of yellow for coloring patches in the environment as we depict in Figure 7.11. We can use PROTEASE with appropriate parameters to achieve such swarm behavior by realizing a collective behavior like that produced by Particle Swarm Optimization (PSO) [Zhang et al., 2015] (video *PROTEASE-NetLogo-PSO* on GitHub and YouTube). For our experiment, we again let all agents in the communication range be covered by $\text{GROUP}_{\text{PROTEASE}}$. Initially, we assume all to be distributed randomly in the environment (cf. Figure 7.11a). We require to extend the list of parameters we need to measure for correctly calculating new trajectories with $\text{CALC}_{\text{PROTEASE}}$ for each agent individually. We extend it with that abstract parameter we are interested in and want the swarm to search for. Thus, we extend C_{PROTEASE}, with the respective capability required for that, e.g., c^p_{PAR} in addition to $c^p_{\text{M-POS}}$ and $c^p_{\text{M-VEL}}$, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{PAR}}, c^p_{\text{M-POS}}, c^p_{\text{M-VEL}}\}$. The calculation function $\text{CALC}_{\text{PROTEASE}}$ thus is described as follows:

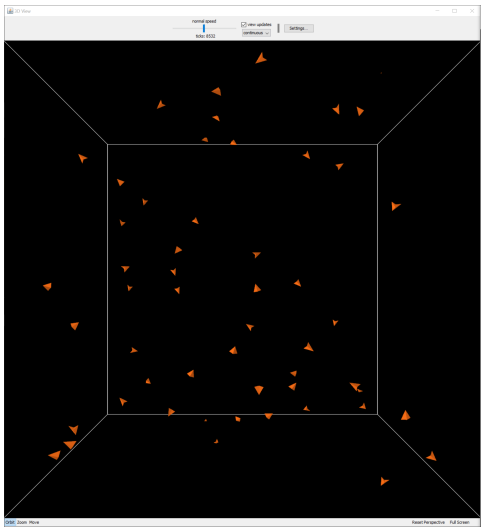
$$\begin{aligned}\text{CALC}_{\text{PROTEASE}} :=& \omega_1 \cdot \text{DIST}(\text{POS}_\alpha, \text{MAX}(\text{M}^{\mathcal{E}_{\text{PROTEASE}}}[\text{SELF}][c^p_{\text{PAR}}], \text{MAX}^{\text{SELF}}))+ \\ & \omega_2 \cdot \text{DIST}(\text{POS}_\alpha, \text{MAX}(\text{M}^{\mathcal{E}_{\text{PROTEASE}}}[*][c^p_{\text{PAR}}], \text{MAX}^{\mathcal{E}_{\text{PROTEASE}}}))+ \\ & \omega_3 \cdot \text{DIST}(\text{POS}_\alpha, \text{RAND}(x, y, z))\end{aligned}$$

as the weighted sum of distance vectors. Each distance vector points from the agent α's current position α_{POS} to the position with the iteratively updated highest measurement of the parameter of interest. The first distance-vector thus points to the agent's locally measured best position, i.e., MAX^{SELF}. The second distance vector points to the maximum covered by agents defined by $\text{GROUP}_{\text{PROTEASE}}$, i.e., $\text{MAX}^{\mathcal{E}_{\text{PROTEASE}}}$. The third distance vector points to a random direction $\text{RAND}(x, y, z)$ we include for exploration purposes (cf. [Zhang et al., 2015] for descriptions of its necessity). We can see how the calculations performed with $\text{CALC}_{\text{PROTEASE}}$ modify the positions and headings of all agents in $\text{GROUP}_{\text{PROTEASE}}$ during the executions in Figure 7.11b, were some agents already determined positions in the environment with a high concentration. In Figure 7.11c, information concerning the position with the highest concentration propagated within the swarm so that most agents already positioned themselves near that point of interest. Similar to the execution of PROTEASE with respective parameters we use for gathering, we can determine a termination function $\text{TERM}_{\text{PROTEASE}}$ executed with parameters producing a PSO-like swarm behavior. By determining whether the diameter of the ensemble is below a threshold x, i.e.,

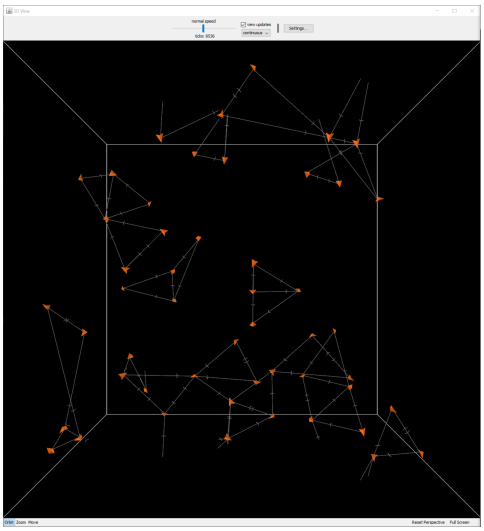
$$\text{TERM}_{\text{PROTEASE}} := \text{DIAM}(\text{M}^{\mathcal{E}_{\text{PROTEASE}}}[*][c^p_{\text{POS}}]) \leq x$$

we know that the desired emergent effect is achieved by the swarm (cf. Figure 7.11d). We can thus again let the result of PROTEASE be defined by the aggregation function $\text{AGG}_{\text{PROTEASE}}$ that determines the centroid from the agents covered by $\text{GROUP}_{\text{PROTEASE}}$ after termination (again, these are all agents in the swarm).

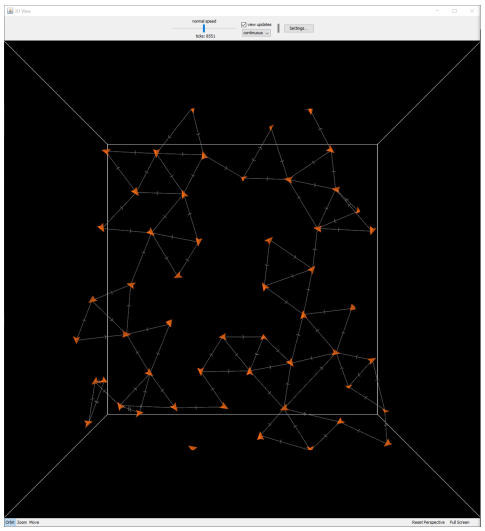
IV) Performing Distributed Surveillance with the Triangle Swarm Behavior When the swarm has finally reached and determined the point of interest, we might want to let the agents collectively survey the area near that point using swarm behavior. We can achieve such Distributed Surveillance of an area of interest using PROTEASE with respective parameters to produce a swarm behavior similar to that described in [Li et al., 2009] that forms a triangle (video *PROTEASE-NetLogo-Triangle* on GitHub and YouTube). We adapt that algorithm to work within a 3D environment as we use for our experiments in NetLogo (and require later



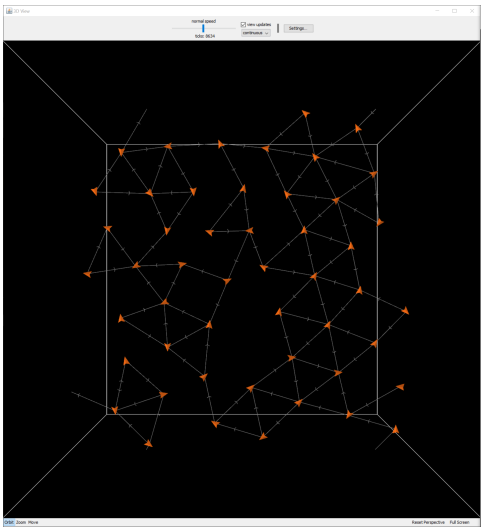
(a) randomized setup for triangle behavior



(b) sub-systems already form triangles



(c) most agents form triangles



(d) all agents form triangles

Figure 7.12: Image exports from the NetLogo simulation environment during executing PROTEASE for achieving a *triangle swarm behavior* in four states from a top down perspective. Agents that try to form triangles with their neighbors are connected by gray arrows.

for applying it to flying ensembles). Like for all other experiments, we first let all agents randomly distributed in the environment (cf. Figure 7.12a). Again, we define $\text{GROUP}_{\text{PROTEASE}}$ to be the group of agents in the communication range. Using our adaptations of the triangle algorithm introduced by [Li et al., 2009] within the calculation function $\text{CALC}_{\text{PROTEASE}}$ we use to determine new trajectories for agents in PROTEASE, we can exploit the emergent effect of a swarm distributing in an area holding a predefined distance s to each other at a given height h. To produce the desired emergent effect, each agent α requires position measurements of its two closest neighbors only, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{M-POS}}\}$. For calculating results with $\text{CALC}_{\text{PROTEASE}}$,

we thus first need to determine the two closest neighbors $\alpha_{1,2}$ of α, i.e.,

$$\begin{aligned}\forall\alpha, \alpha \neq \alpha_i \neq \alpha_1 \neq \alpha 2 :& \\ \forall\alpha_i \in \mathcal{E}_{\text{PROTEASE}} :&\text{DIST}(\alpha, \alpha_i) \geq \text{DIST}(\alpha, \alpha_1) \wedge \\ \forall\alpha_i \in \mathcal{E}_{\text{PROTEASE}} \setminus \alpha_1 :&\text{DIST}(\alpha, \alpha_i) \geq \text{DIST}(\alpha, \alpha_2)\end{aligned}$$

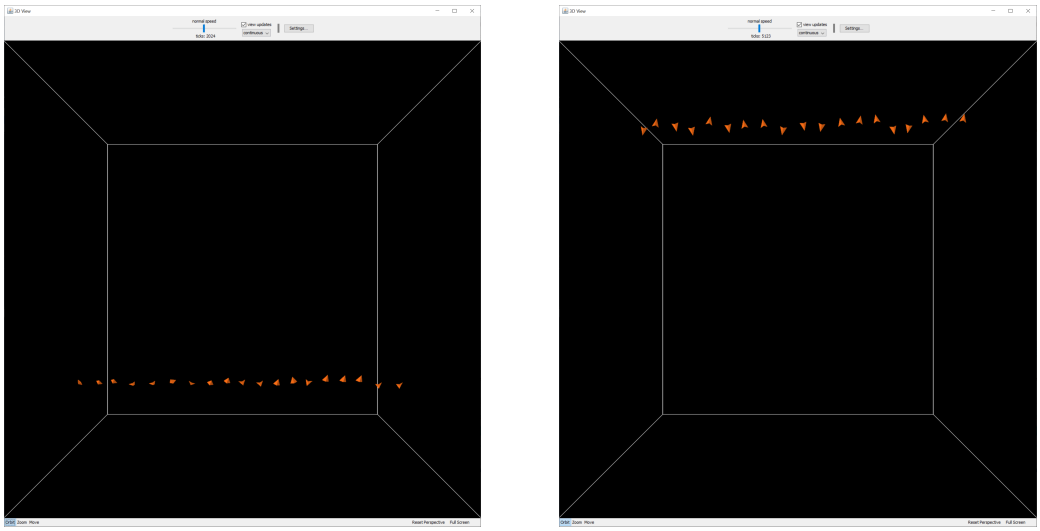
We then calculate the the centroid $\text{CENTR}(\alpha_1, \alpha_2)$ between α_1 and α_2 and determine the distance vector pointing from α to the closest intersection point of the plane at height h (defined parallel to ground level) and the circle around the centroid with radius $\sqrt{3} \cdot \frac{s}{2}$ (being perpendicular to the straight defined by α_1 and α_2) as the goal position of α. We see how this calculation adapts the positions of agents in the environment in Figure 7.12b and continuously improves the covered area in Figures 7.12c and 7.12d. While we can define a function $\text{TERM}_{\text{PROTEASE}}$ for terminating the execution of PROTEASE without using additional measurements, e.g., if all distances between closest neighbors only vary marginally for all agents in the swarm, we do not want to specify such in the case of continuous surveillance. If we reached such a state (cf. Figure 7.12d), we instead want to continue with the surveillance until a specific event happens. This can either be user input like executing PROTEASE to produce a guided boiding swarm behavior, or we extend the measurements agents perform with a specific parameter of interest indicating a termination event. If we have defined such, we thus can terminate the execution of PROTEASE, e.g., when one agent measured such event and likewise use the event's position as the result of PROTEASE by defining a respective aggregation function $\text{AGG}_{\text{PROTEASE}}$. As we stated above, investigating in $\text{TERM}_{\text{PROTEASE}}$ and $\text{AGG}_{\text{PROTEASE}}$ is more straightforward when embedding them in Ensemble Programs as we do it in Section 7.6.2.

7.6.1.3 Further Applications of PROTEASE

Besides applying PROTEASE for the case study of *Dealing with Gas Accidents*, we can also use it for realizing swarm behavior that can be beneficial in other scenarios. We thus give an excerpt of such applications in the following. Thereby, we briefly describe how to parametrize PROTEASE for achieving the individual swarm behavior and in which applications the respective swarm behavior can be of use. For the sake of brevity, we neglect parameters $\text{AGG}_{\text{PROTEASE}}$ and $\text{TERM}_{\text{PROTEASE}}$ for the following descriptions, as we do not analyze them in our NetLogo experiments.

Measuring in Line Formation: The Line of Fliers Having our case study on *Innovative Measurement Methods* in mind and recapitulating the evaluations we performed therein during the field-experiments involving real hardware in the ScaleX 2016 campaign (cf. Section 3.4.2), we might want to apply swarm behavior for achieving a flying formation as an emergent effect similar to that we applied there. Sousselier et al. [2012] describe such swarm behavior for UUVs we can adapt to work with PROTEASE (video *PROTEASE-NetLogo-Line-of-Fliers* on GitHub and YouTube). Like for all other swarm behavior we simulate in NetLogo in this section, we also define $\text{GROUP}_{\text{PROTEASE}}$ to cover all other agents in communication range unless we only require information on the position of the closest neighbor in $\text{CALC}_{\text{PROTEASE}}$ in addition to information on the 6-dimensional[5] orientation of the (virtual) line, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{M-POS}}, c^p_{\text{REF}}\}$. Similar to the approach of Reynolds [1987], we can then describe $\text{CALC}_{\text{PROTEASE}}$ as the combination of

[5]We describe this position as transformation, combining 3-dimensional position and orientation.

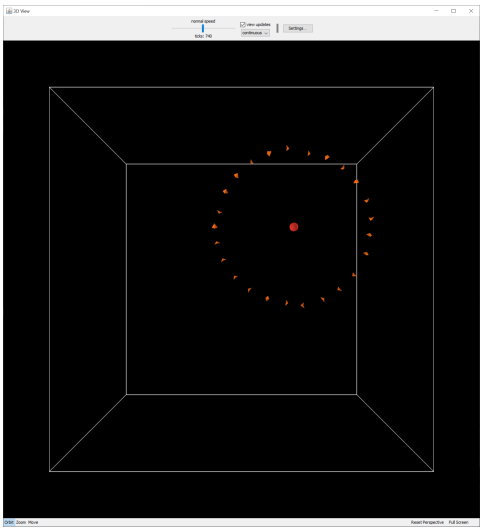


(a) initial position (b) following the leader

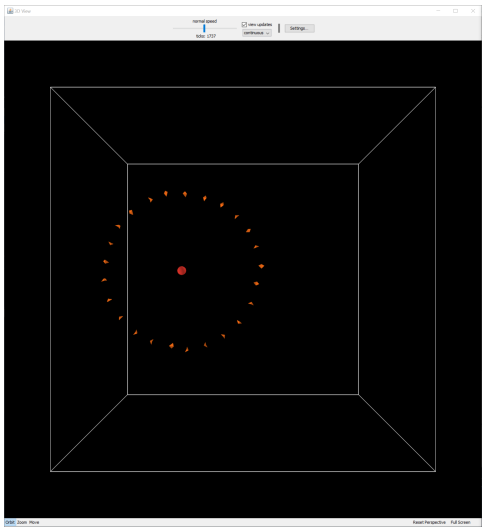
Figure 7.13: Image exports from the NetLogo simulation environment during executing PROTEASE for achieving a *user-controllable line formation swarm behavior* from a top down perspective.

three urges. One urge points from the respective agent's current position towards the closest position on the virtual line. A second urge points away from the closest neighbor covered by $\text{GROUP}_{\text{PROTEASE}}$. A third urge points towards the center of the line. Combining those urges appropriately generates a formation like that we depict in Figure 7.13a. Again, we can integrate a user-controlled leader agent into the swarm, enabling the user to steer the swarm as required, e.g., move the swarm in a certain height along a route like we depict in Figure 7.13b (cf. also our ScaleX experiment we describe in Section 3.4.2.1).

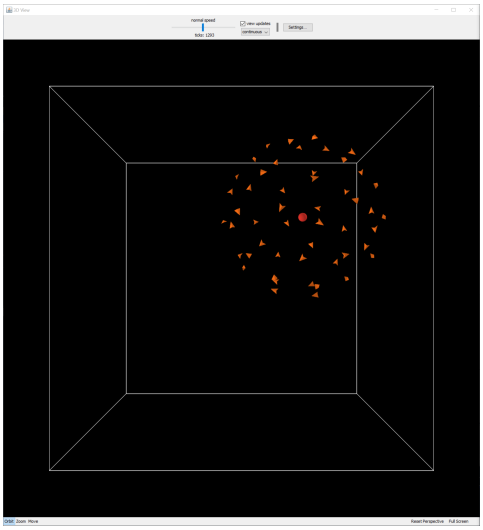
Measuring in Ring and Ball Formation: The Ring and Ball-of-Fliers Another swarm behavior we can produce using PROTEASE with appropriate parameters is that of forming a ring or a ball (cf. Figure 7.14 and the videos *PROTEASE-NetLogo-Ring-of-Fliers* and *PROTEASE-NetLogo-Ball-of-Fliers* on GitHub and YouTube). Recapitulating our experiment we performed using real hardware during the ScaleX 2015 campaign where we required multiple agents to form a measuring tower, we see that such formation flight can be beneficial for flying ensembles when scaling the number of participating agents. We let $\text{GROUP}_{\text{PROTEASE}}$ describe the set of agents within the communication range. For producing the desired emergent effect, we must provide each agent with information on its own position and the 6-dimensional orientation of the reference point describing the center of the circle or ball respectively combined with a constant describing the desired radius for the formation, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{M-POS}}, c^p_{\text{REF}}\}$. Again, we can describe $\text{CALC}_{\text{PROTEASE}}$ as the combination of different urges, each described by a vector pointing into the desired moving direction from the respective agent. A first urge points towards the measured position describing the center of the ring or ball. A second urge points away from the closest neighbor covered by $\text{GROUP}_{\text{PROTEASE}}$. While these two urges already generate a ball formation, we need to include a third urge to generate a circle formation, holding the agents in the swarm in the plan described by the reference point's orientation. Thus, that



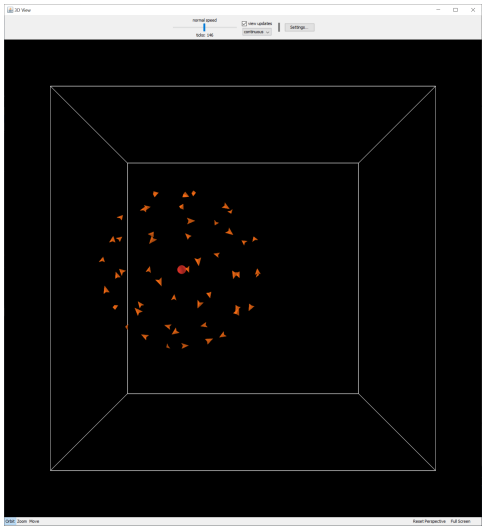
(a) Ring-of-Fliers: initial position



(b) Ring-of-Fliers: following the leader



(c) Ball-of-Fliers: initial position

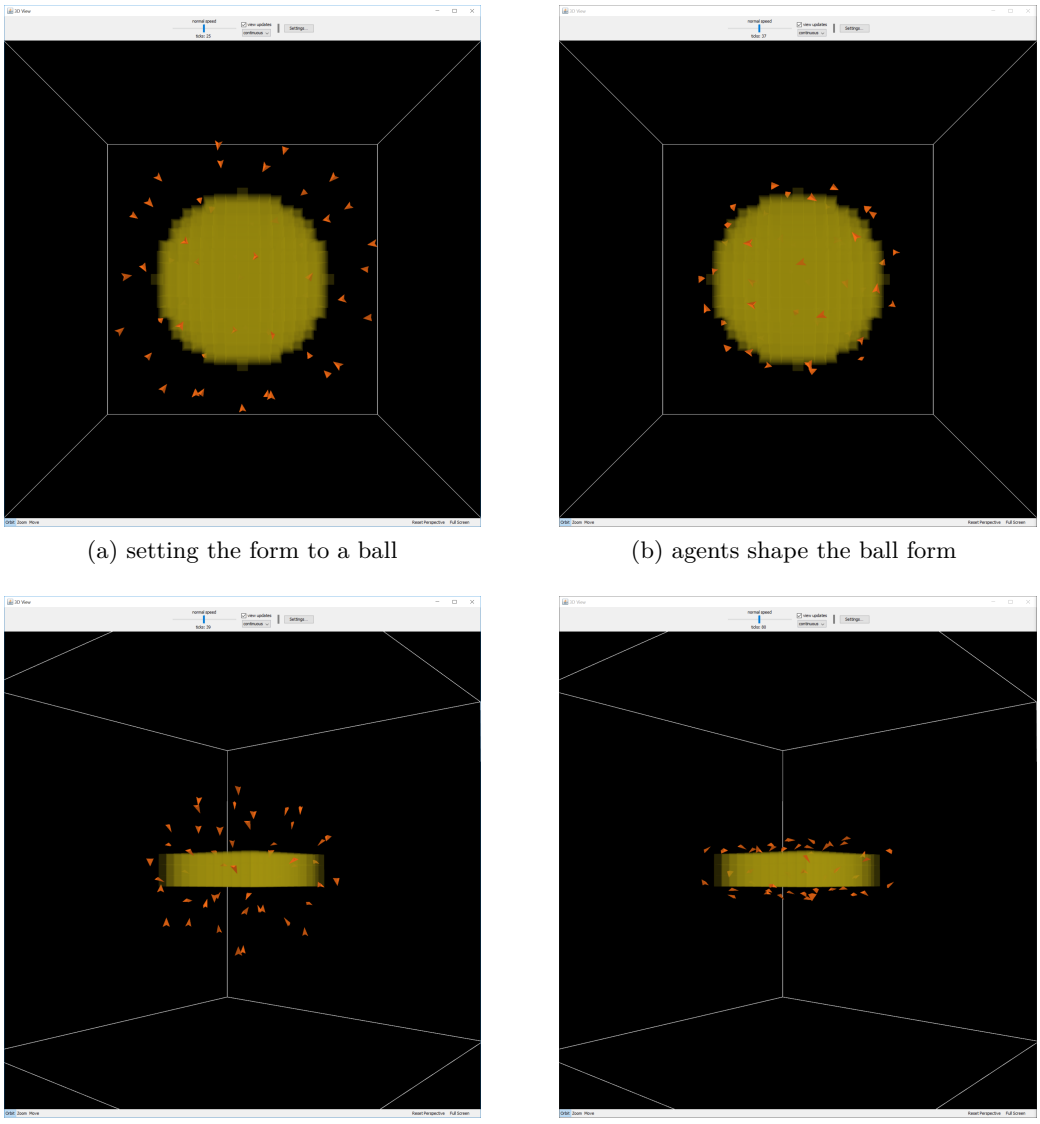


(d) Ball-of-Fliers: following the leader

Figure 7.14: Image exports from the NetLogo simulation environment executing PROTEASE for achieving a *user-controllable Ring-of-Fliers swarm behavior* in Figures 7.14a and 7.14b and a *user-controllable Ball-of-Fliers swarm behavior* in Figures 7.14c and 7.14d. The guiding user-controlled agent is represented by a red ball.

third urge points towards the closest position of this plane. We see how PROTEASE produces the desired formation in Figure 7.14a for the ring and Figure 7.14c for the ball formation. Like for using PROTEASE producing the line formation, we can also move the reference point in space, commanding the swarm to collectively follow it and reform in the desired shape (cf. Figures 7.14b and 7.14d).

Filling and Shaping Forms with an Adapted Potential Field Algorithm Besides using PROTEASE to generate swarm behavior for case studies as we describe them in this thesis,



(a) setting the form to a ball

(b) agents shape the ball form

(c) changing the form to a cuboid

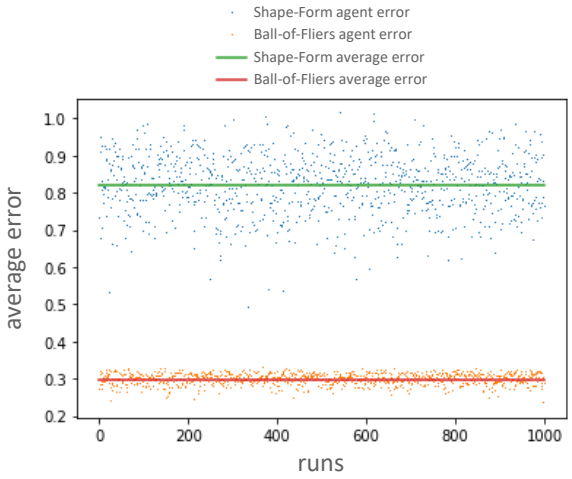
(d) agents fill the cuboid form

Figure 7.15: Image exports from the NetLogo simulation environment during executing PROTEASE for a *Shape-Form swarm behavior*. The yellow-colored environment indicates the form to fill by the swarm.

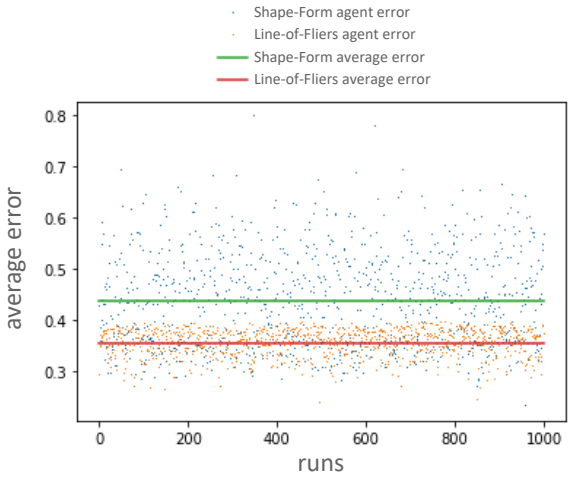
we can also use it to generate such emergent effect in other applications for flying ensembles (*Shape-Form* on GitHub and YouTube). In Section 7.2, we introduced the work performed by Intel [2021] for generating light shows involving large scale ensembles. We can use PROTEASE to produce similar formations. Therefore, we define $\text{GROUP}_{\text{PROTEASE}}$ to be the set of agents in the communication range. In $\text{CALC}_{\text{PROTEASE}}$, each agent requires information on its position and information on the position of a 3-dimensional body in space, combined with a description of the body's shape, i.e., $C_{\text{PROTEASE}} := \{c^p_{\text{M-POS}}, c^p_{\text{REF}}\}$. We can then describe $\text{CALC}_{\text{PROTEASE}}$ as the combination of different urges described by direction vectors pointing from the agent's position towards the desired destination. For the swarm behavior for filling a form, we thus require

the first urge to point towards the closest position in space inside the form. A second urge points away from the closest neighbor covered by $\text{GROUP}_{\text{PROTEASE}}$. While the combination of these urges already causes the swarm to distribute within the defined form equally, we might want to let the swarm only shape the body's form, i.e., avoid moving into the space inside of the form. If so, we can add a third urge that points away from the center of the form so that positions within the form are undesirable to enter for an agent in the swarm. We can see how PROTEASE performs in achieving such formations in Figure 7.15. There, we visualize the form with yellow-colored patches. Further, we depict the more complicated case of shaping only. We first let all agents distribute randomly in the environment. Then, we let the agents execute PROTEASE using the respective parameters to shape the ball form (cf. Figure 7.15a). We then modify the form to a cuboid so that all agents need to reposition themselves on the new body's shape (cf. Figure 7.15c). We see how the swarm is finally achieving the desired behavior in Figure 7.15d. Further demonstrations of the possibilities PROTEASE offers for shaping and filling forms can be found in Appendix B.1, including also forms similar to that we can achieve with the Line-of-Fliers algorithm. While the forms we support currently are still limited compared to that of Intel [2021], we potentially can extend the specific parameters of PROTEASE to allow for much more complex forms, e.g., by adding support for combined forms consisting of multiple simple forms as we show them in our figures. While this topic is still purpose of future work, we see high potential for exploiting the benefits of the paradigm of self-organization, i.e., scalability and robustness, for this type of application.

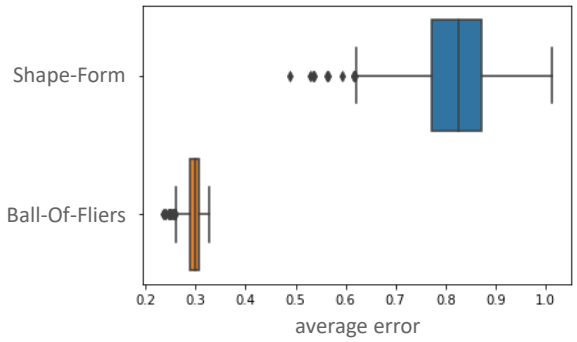
Comparing Different Parametrization of PROTEASE Producing Similar Effects Because we now have two different sets of parameters for PROTEASE at hand generating very similar emergent effects, i.e., lines and balls (cf. the Line-of-Fliers and the Ball-of-Fliers algorithm), we compare their results concerning the quality they achieve for forming the desired form. Results from these evaluations partially originate from [Rall, 2019]. For all instantiations of PROTEASE using different parameters, we use a metric measuring the euclidean distance to the desired form for each agent in the swarm, i.e., measuring the error for each agent. Figure 7.16 depicts the results of 1000 runs we performed for both line-forming instantiations of PROTEASE. In the scatter-plot in Figures 7.16a and 7.16b, we depict the lowest average error a swarm consisting of 40 agents could collectively achieve for each run within 200 iterations we performed within each of the 1000 runs of the experiment. Comparing the average line for all 1000 runs, we can see that the Line-Of-Fliers algorithm outperforms the Shape-Forming algorithm significantly ($p = 1.87 + 10^{142}$), and the Ball-Of-Fliers algorithm outperforms the Shape-Forming algorithm significantly ($p = 0$). We can see our findings cross-validated when comparing the box-plots depicting the distribution of results for both comparisons in Figures 7.16c and 7.16d. While the Line-Of-Fliers and the Shape-Forming algorithm still show some overlap in results in their upper quartiles (cf. Figure 7.16d), there is no such overlap for the Ball-Of-Fliers and the Shape-Forming algorithm (cf. Figure 7.16c). Thus, we can conclude that the Shape-Forming algorithm loses accuracy due to its generality concerning the shapes it can form with a swarm. If we require precise line or ball formations in a specific application, we thus recommend using the specialized parametrization for PROTEASE (Ball-Of-Flier and Line-Of-Flier). If, instead, precision is not of topmost interests and versatility in forms we want to build with the swarm is of more interests, e.g., in light shows like that demonstrated by Intel [2021], we recommend using the Shape-Forming algorithm instead.



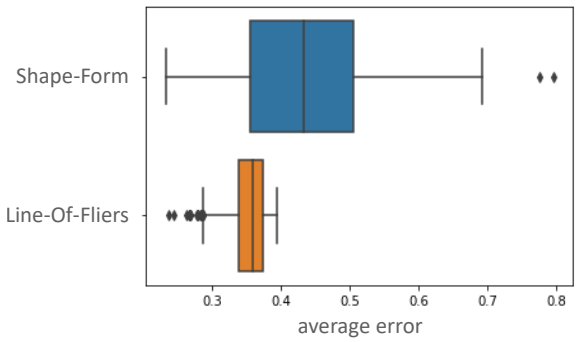
(a) scatter plot for ball-like formations



(b) scatter plot line-like formations



(c) box plot for ball-like formations



(d) box plot for line-like formations

Figure 7.16: Evaluation results comparing different parametrization of PROTEASE used for building line-like and ball-like formations, figure adapted from [Rall, 2019].

7.6.2 Evaluating the Coordinated Execution of Ensemble Programs

We evaluate on the approach of coordinated execution of Ensemble Programs we introduced in Section 7.4 using our prototypical reference implementation of Multipotent Systems. In our implementation, we let every agent $\alpha \in \mathcal{A}_{\mathcal{MS}}$ implement the EPU program and PFC from Section 7.4 extended with the additional procedures we require for executing Collective Capabilities (cf. Section 7.5). For each different aspect, we provide video materials on GitHub and YouTube[6].

7.6.2.1 Planning-Agent-Groups

In the first set of evaluations, we investigate the different Planning-Agent-Groups we introduced in Section 4.4.4 and created examples for in the evaluations on the expressiveness of MAPLE in Section 4.5.1.1. In the video materials we provide on GitHub and YouTube, we see how the agents perform the respectively commanded executions from Figure 4.9 we pro-

[6] `https://github.com/isse-augsburg/ensemble-programming` or `https://github.com/kosakoliver/ensemble-programming` or `https://www.youtube.com/user/ISSELabs`

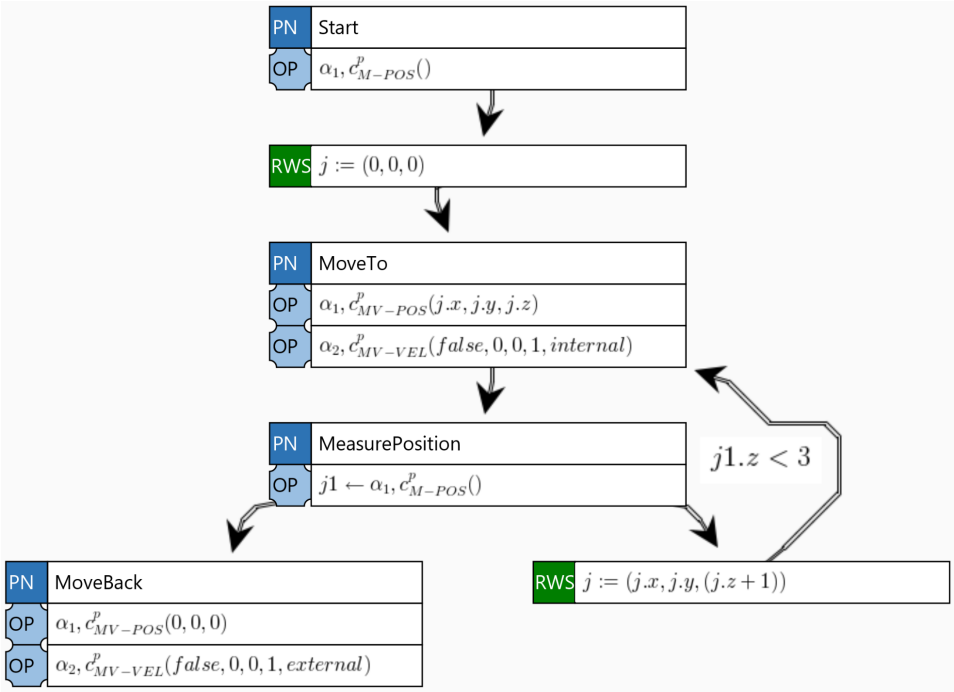


Figure 7.17: Parallel version of Figure 4.10 involving a second agent that executes a non-self-terminating capability ($c^p_{\text{MV-VEL}}$) with different termination levels (one internal, i.e., $s = \bot, e = \top$, one external, i.e., $s = \bot, e = \top$, cf. Section 4.4.2). Exported from our HTN designer

grammed using our user-frontend for designing HTN in MAPLE. We provide one video for each possible Planning-Agent-Group, i.e., the Any-Agent (video *MAPLE-Any-Agent* on GitHub and YouTube, cf. Figure 4.9a), the All-Agent (video *MAPLE-All-Agent* on GitHub and YouTube, cf. Figure 4.9b), the Swarm-Agent (video *MAPLE-Swarm-Agent* on GitHub and YouTube, cf. Figure 4.9c), and the Set-Agent (video *MAPLE-Set-Agent* on GitHub and YouTube, cf. Figure 4.9d), for demonstrating its isolated functionality.

7.6.2.2 Control Structures and World State Modifications at Run-Time

In a second set of evaluations we investigate the different control structures we introduced in Section 4.4.5 and created examples for in the evaluations on the expressiveness of MAPLE in Section 4.5.1.3. In the video materials we provide on GitHub and YouTube, we see how agents execute the plans derived from the HTNs we provide in Figure 4.10 to demonstrate a simple sequential execution of one agent first (video *MAPLE-Sequential-Execution* on GitHub and YouTube). To demonstrate parallelism, we introduce another agent in the plan (cf. Figure 7.17) and provide a video showcasing its execution (video *MAPLE-Parallel-Execution* on GitHub and YouTube). We thereby create different combinations of self-finishing and non-self-finishing behavior we require from both agents.

In another video (*MAPLE-Repeated-and-Conditional-Execution* on GitHub and YouTube), we demonstrate how repeated and conditional executions of Ensemble Programs can be executed with our system. Therefore, we let our system execute the plan from Figure 4.12. This example further demonstrates how world state modifications performed at run-time have a

dynamic influence on the execution of the respective Ensemble Programs.

7.6.2.3 World State Modification at Planning Time and Replanning

In a third set of evaluations we investigate the different ways for modifying variables in WS and using them for creating new plans during the execution of Ensemble Programs we introduced in Sections 4.4.3 and 4.4.6 and created examples for in the evaluation of the expressiveness of MAPLE in Section 4.5.1.4. In the video *MAPLE-Replanning-Execution* we provide on GitHub and YouTube, we see how the agents in the system execute the plans derived from the HTN we depict in Figure 4.14. After a first execution of the plan from Figure 4.14b, a replanning is triggered autonomously by the executing ensemble that subsequently broadcasts the plan from Figure 4.14c that is executed by another ensemble.

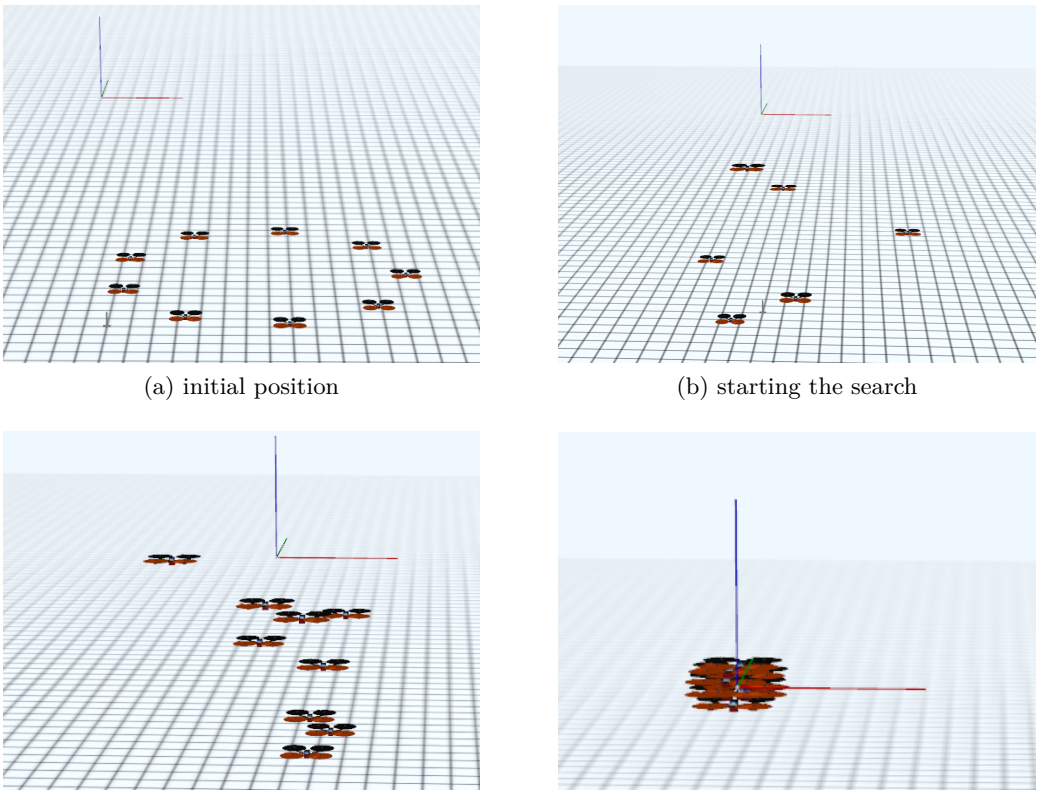
7.6.2.4 Collective Capability Encapsulating PROTEASE

In a fourth set of evaluations we investigate the goal oriented integration of PROTEASE into a Collective Capability c^v_{PROTEASE} like we introduced in Section 7.5. That way, we can enable the execution of c^v_{PROTEASE} as particular form of a virtual capability as we describe them in Section 3.3 and visualize in Figure 3.27. To allow for different parameters c^v_{PROTEASE}, we offer the possibility to change $\text{AGG}_{\text{PROTEASE}}$, $\text{TERM}_{\text{PROTEASE}}$, and $\text{CALC}_{\text{PROTEASE}}$ explicitly within our user interface for defining plans and let the system autonomously determine on the respectively included agents in $\text{GROUP}_{\text{PROTEASE}}$ by the mechanisms for Ensemble Formation we proposed in Chapter 5. The way we integrate those concepts into our reference implementation (cf. snipped of a class diagram describing the relevant concepts in Appendix B.2) allows for the easy extension of our system with new swarm behavior derived by increasing the set of possible parameters we provide for executing PROTEASE.

We assume to have sufficiently equipped ensembles $\mathcal{E}_{\text{PROTEASE}}$ at hand concerning the set of required capabilities C_{PROTEASE} necessary for that concrete instantiation. In our video materials on GitHub and YouTube[7], we evaluate the effect of changing the different parameters of PROTEASE.

Changing the Calculation Function of PROTEASE Modifying $\text{CALC}_{\text{PROTEASE}}$ has the biggest impact on the execution of PROTEASE. In the individual videos *PROTEASE-Calculator-PSO*, *PROTEASE-Calculator-Boiding*, *PROTEASE-Calculator-Triangle*, *PROTEASE-Calculator-Potential-Field*, *PROTEASE-Calculator-Line*, *PROTEASE-Calculator-Ring-of-Fliers*, *PROTEASE-Calculator-Gathering*, and the video *PROTEASE-Calculator-Ball-of-Fliers* we provide on GitHub and YouTube, we demonstrate the collective effect we achieve when changing this parameter. For all different types of $\text{CALC}_{\text{PROTEASE}}$, we further provide a *user-controlled* version where we integrate a special agent into the group $\text{GROUP}_{\text{PROTEASE}}$ allowing the user to lead the collective (see respective **-User* videos for differently set $\text{CALC}_{\text{PROTEASE}}$ on GitHub and YouTube). We represent that special agent as a simulated version of the pointing device we used for our evaluations on Ensemble Formation in Section 5.5.2. We designed the set of calculation functions we provide to be easily extendible. In Figure 7.18 we show image exports from the simplified visualization we integrate our prototypical reference implementation of Multipotent System with. We use this for demonstrating an exemplary execution of PROTEASE executing

[7] https://github.com/isse-augsburg/ensemble-programming or https://github.com/kosakoliver/ensemble-programming or https://www.youtube.com/user/ISSELabs



(a) initial position (b) starting the search

(c) converging towards the goal (d) stabilizing at the goal

Figure 7.18: Execution of PROTEASE encapsulated in the virtual capability c^v_{PROTEASE}. Parameter $\text{CALC}_{\text{PROTEASE}}$ is configured to produce a PSO-like swarm behavior. Starting from an initial position Figure 7.18a over the search phase Figure 7.18b and Figure 7.18c until the swarm stabilizes at the position of highest concentration of the simulated gas g Figure 7.18d, we mark with a coordinate system in the visualization.

a PSO-like swarm behavior with a swarm consisting of nine agents. We further use this exemplary execution to evaluate the self-finishing property of c^v_{PROTEASE} using appropriately set $\text{TERM}_{\text{PROTEASE}}$ in the following.

Changing the Termination Function of PROTEASE Modifying $\text{TERM}_{\text{PROTEASE}}$ changes how the execution of different swarm behavior terminates. We support different termination functions, including such working autonomously, i.e., self-finishing, and such requiring external coordination from the user, i.e., non-self-finishing. We designed the set of termination functions we provide to be extendible in the future.

In our video *PROTEASE-Termination-Centroid* on GitHub and YouTube, we let the system execute PROTEASE having $\text{CALC}_{\text{PROTEASE}}$ configured to produce a gathering swarm behavior. The execution terminates autonomously when the agents collectively executing the swarm behavior determine that distances to all other agents covered by $\text{GROUP}_{\text{PROTEASE}}$ are below a predefined threshold of 0.2 meters. While this threshold value is valid in our simulation environment, we need to adapt it when executing c^v_{PROTEASE} in a setting involving real hardware to avoid collisions and respect possible minimal reachable distances introduced by a collision

avoidance mechanism. Thus, the behavior in our reference implementation using c^v_{PROTEASE} is very similar to that we describe in Section 7.6.1.2 when using our NetLogo simulation environment.

In our video *PROTEASE-Termination-Unknown-Measurement* on GitHub and YouTube, we let the system execute PROTEASE having $\text{CALC}_{\text{PROTEASE}}$ configured to produce a triangle formation swarm behavior controlled with the user-controlled agent. The execution terminates autonomously when one of the agents determines the occurrence of a special event. To define that event, the user can set a specific other capability in $\text{CALC}_{\text{PROTEASE}}$ to decide for that event. In our example, we let the swarm terminate its execution of PROTEASE if a measurement of the simulated gas g achieved by $c^p_{\text{M-GASG}}$ is greater than a specific threshold.

In our video *PROTEASE-Termination-User* on GitHub and YouTube, we let the system execute PROTEASE having $\text{CALC}_{\text{PROTEASE}}$ configured to produce a boiding swarm behavior where we integrate the user-controlled agent for steering the swarm. The execution terminates triggered by the user by interacting with the executing ensemble through the input possibilities we provide in our user front end.

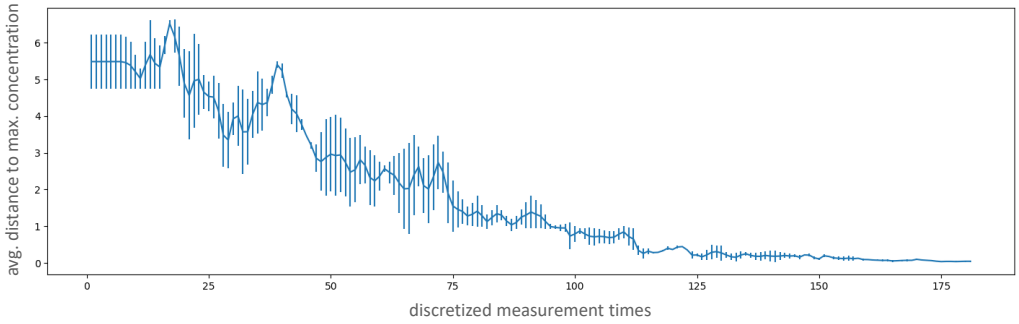
We empirical evaluate on the feasibility of such termination functions and present the results in Figure 7.19. We plot the results of two exemplary runs of PROTEASE encapsulated in a virtual capability c^v_{PROTEASE} having the parameter $\text{CALC}_{\text{PROTEASE}}$ configured to produce a PSO-like swarm behavior (cf. Figures 7.19a and 7.19b). There, we see the average distance to the maximum concentration of the simulated GAS G. We calculated the average for all agents in the swarm. We see, the measured value drops close to zero and standard deviation continuously over time, i.e., when agents collectively determine the position of highest concentration Thus, we can apply a parameter $\text{TERM}_{\text{PROTEASE}}$ for executing PROTEASE encapsulated in c^v_{PROTEASE} using a centroid-based termination function to actually terminate the capability's execution autonomously.

We achieve similar results for another execution of PROTEASE encapsulated in c^v_{PROTEASE} for producing a *potential-field-based swarm behavior* for achieving an equal distribution with a specific area. In Figures 7.19c and 7.19d we plot the results of two exemplary runs. We measure the average distance between agents in the swarm. For every agent, we calculate this value regarding its distance to its closest neighbor in the swarm. We see that from an initially low average distance between agents, e.g., after executing a PSO swarm behavior, also this value stabilizes over time. The value converges a specific constant value depending on the number of agents and the dimensions of the area to fill. Thus, also for this parametrization of PROTEASE we can define a termination function in its parameter $\text{TERM}_{\text{PROTEASE}}$. The execution of PROTEASE then can terminate if changes in the value calculated with the metric we define does not further change significantly. which we can use for autonomous termination of the capability c^v_{PROTEASE}.

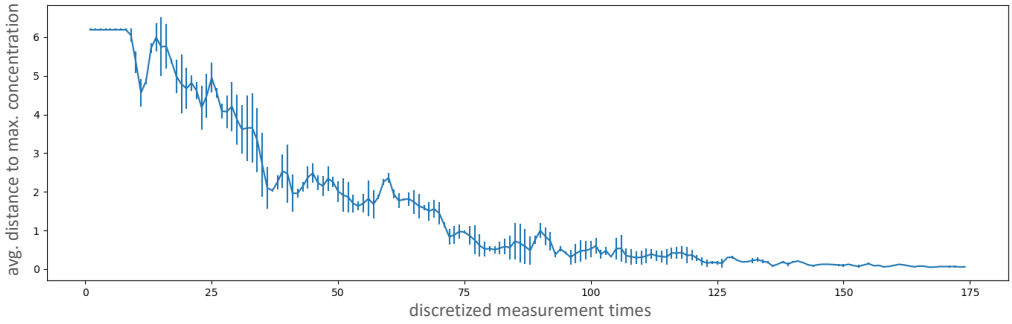
Thus, by defining an appropriately configured parameter $\text{TERM}_{\text{PROTEASE}}$ for a specific chosen parameter $\text{CALC}_{\text{PROTEASE}}$, we can let the execution of PROTEASE terminate autonomously when we encapsulate it in c^v_{PROTEASE}.

Changing the Aggregation Function of PROTEASE Modifying $\text{AGG}_{\text{PROTEASE}}$ does not change the respectively executed swarm behavior's emergent effect, but how we quantify it for its further usage. We include different possibilities enabling such quantification and design them to be extendible in the future.

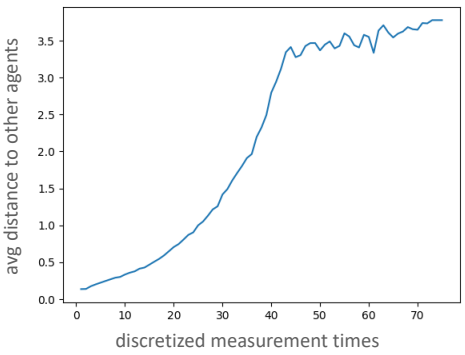
In our video *PROTEASE-Aggregation-Centroid* on GitHub and YouTube, we let the system



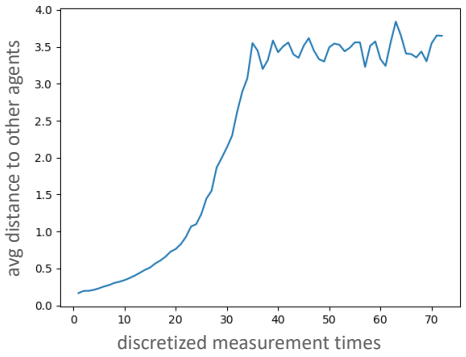
(a) PSO swarm behavior, 1st run



(b) PSO swarm behavior, 2nd run



(c) potential-field swarm behavior, 1st run



(d) potential-field swarm behavior, 2nd run

Figure 7.19: Results originating from [Bohn, 2018] derived during different exemplary executions of PROTEASE encapsulated in the virtual capability c^v_{PROTEASE}. In Figures 7.19a and 7.19b, we analyze exemplary executions of c^v_{PROTEASE}, each using the parameter $\text{CALC}_{\text{PROTEASE}}$ set to produce a *PSO-like swarm behavior*. In Figures 7.19c and 7.19d, we analyze exemplary executions of c^v_{PROTEASE}, each using the parameter $\text{CALC}_{\text{PROTEASE}}$ set to produce a *potential-field-based separation swarm behavior*.

execute PROTEASE with $\text{CALC}_{\text{PROTEASE}}$ configured to produce a PSO-like swarm behavior with $\text{TERM}_{\text{PROTEASE}}$ configured to terminate after finding the highest concentration of the simulated gas g. We then let PROTEASE aggregate the result of this execution using $\text{AGG}_{\text{PROTEASE}}$ set to a function calculating the centroid from all agents in the swarm in that final state. The ensemble executing the swarm behavior then uses that result to establish a predefined tower formation at the determined position we can use in a more extensive example, e.g., to investigate the gas's dissemination.

In our video *PROTEASE-Aggregation-Unknown-Measurement* on GitHub and YouTube, we let the system execute PROTEASE with $\text{CALC}_{\text{PROTEASE}}$ configured to produce a user-controlled boiding swarm behavior which we terminate autonomously with $\text{TERM}_{\text{PROTEASE}}$ after the swarm determines a new measurement of the simulated gas g that is higher than a certain threshold. We define the result of executing PROTEASE as the position where this new measurement was derived in $\text{AGG}_{\text{PROTEASE}}$. Again, we use the result to establish a formation around that position, this time investigating the horizontal dissemination of gas g.

7.6.2.5 Evaluating of the External Interface: The Protelis Capability

In this section, we evaluate the feasibility of integrating an external programming language as Collective Capability. Therefore, we provide an instance of c^v_{EXT} encapsulating the logic required to execute programs generated with the Protelis Aggregate Programming approach of Pianini et al. [2015], i.e., a further Collective Capability c^v_{PROT}. We perform our evaluations using our prototypical reference implementation of Multipotent Systems we introduce in Section 3.3. To validate the concepts we introduced in Section 7.5.4, we give proof of concepts concerning the relevant parts executing an external capability. These concepts are the *communication* between participating agents, commanding the *execution* and making use of the *results* of capabilities running on the host system, and ensuring *self-termination* of the external capability, if necessary. According to [Pianini et al., 2015], for communication between entities, Protelis requires a network manager. With c^v_{PROT} we implement such (cf. Line 7 in Algorithm 10). We can validate its functionality with the minimal example of a Protelis program we give in Listing 7.1 that counts all members of the ensemble using the nbr construct in Line 2 of Listing 7.1. The example showcases the ability of communication between agents executing c^v_{PROT}. In the Protelis program in Listing 7.3, we demonstrate how external capabilities can define required access to other capabilities of the host system (implemented in JAVA) using the self construct of Protelis for measuring temperature (Line 12 in Listing 7.3). In Line 5-8 of Listing 7.3, we access the knowledge base of our architecture by importing the ParamFactory (Line 3 in Listing 7.3). We use this knowledge base for loading the correct format of the necessary parameters for the measure temperature capability. For achieving this, we make use of the JAVA Reflection API. With self.request (Line 9 in Listing 7.3), we define the request the external capability has concerning the execution of physical capabilities (Line 3 in Algorithm 10) whose result we return in Line 10 of Listing 7.3 when it is available. To avoid blocking the Protelis program's execution when it requests a capability execution, we implement the data interface to the host system as a reload cache. To validate the correct program flow and validate correct self-termination of c^v_{PROT}, in the Protelis program we give in Listing 7.1 we let each member of the ensemble iterate a counter (Line 6 in Listing 7.2). Because there is no access to physical capabilities included in the program, the execution of each instance terminates after 10 iterations and accordingly notifies the encapsulating external capability c^v_{PROT} with TERM_{EXT} evaluating TRUE when it finally reaches self.terminate() in Line 4 in Listing 7.2.

```
1 module count_neighbors
2 let num_of_neighbors = sumHood(nbr(1))
3 num_of_neighbors
```

Listing 7.1: testing the communication between agents

```
1 module term_after_iterations
2 def iterations() = rep(x <- 0) { x + 1 }
3 def term_after(x) =
4     if(iterations() > x) { self.term() }
5     else { iterations() }
6 terminate_after(10)
```

Listing 7.2: testing the self-termination on the host system

```
1%\begin{lstlisting}[language=Protelis]
2 module measure_temp
3 import ParamFactory.get;
4 def measure_temp() {
5   let cap_type = self.getType("temp")
6   let measurement_param = get(cap_type)
7   let param = measurement_param.get()
8   param.set("measureOnce", true)
9   let temp = self.request(param, cap_type)
10  temp
11 }
12 measure_temp()
```

Listing 7.3: testing access to the capabilities of the host system

Figure 7.20: Minimal Protelis programs in Listing 7.1, Listing 7.2, and Listing 7.3 we use for validating the concept of an c^v_{EXT} (cf. videos *Protelis-Measure-Temperature-Test, Protelis-Termination-Test*, and *Protelis-Count-Neighbors-Test* on GitHub and YouTube).

Thus, we demonstrate the feasibility of integrating an interface between Protelis, a host system (our prototypical reference implementation of Multipotent System) with a specific Collective Capability c^v_{PROT} as a proof of concepts for our concept from Section 7.5.4. Like for our other evaluations in this section, we provide video materials for demonstration purposes on GitHub and YouTube[8] (videos *Protelis-Count-Neighbors-Test*, *Protelis-Measure-Temperature-Test*, and *Protelis-Termination-Test*). The integration of c^v_{PROT} currently is limited to only execute one Protelis program per agent in parallel and relies on capabilities provided by the host system to terminate on their own, i.e., capabilities must be self-finishing (cf. Section 7.5.2).

[8] `https://github.com/isse-augsburg/ensemble-programming` or `https://github.com/kosakoliver/ensemble-programming` or `https://www.youtube.com/user/ISSELabs`

7.6.2.6 Combined Execution of MAPLE Features

In our video *MAPLE-Combined-Execution* we show the execution of an exemplary HTN combining the possibilities we have for defining Ensemble Programs in MAPLE using our prototypical reference implementation. We provide the HTN in Appendix C. The HTN combines sequential executions with logical and physical parallelism defined by Operators included in the respective Primitive-Nodes, uses repeated and conditional execution based on variables defined in the Worldstate and modified during run-time using Runtime-Worldstate-Modification-Nodes, evaluates conditions defined in Planning-Time-Worldstate-Modification-Nodes during planning, triggers replanning using Replanning-Nodes, involves c^v_{PROTEASE} as an concrete instance of a Collective Capability and uses respective results for subsequent executions.

7.7 Future Research Directions and Preliminary Results

We want to point out a possible future research direction relevant for making the self-organized ensemble program execution more robust against failures in non-Collective Capabilities. We, therefore, refer to the different roles we defined in the reference architecture for Multipotent System in Section 3.2. We provide some preliminary results here.

7.7.1 Robust Execution Besides Collective Capabilities

While we already introduced robustness to program execution with measures of self-organization we provide through the execution of different goal-oriented swarm behavior enabled by PROTEASE which we encapsulate in a virtual capability c^v_{PROTEASE}, we did not yet focus the robust execution of ensemble programs during the execution of non-Collective Capabilities. In the following we investigate in this topic by first analyzing possible failures that can occur while executing ensemble programs including our analysis for possible countermeasures we can apply in a self-organized manner. Second, we provide a concept for handling failures in Multipotent Systems, whose feasibility we support with some preliminary results.

7.7.1.1 Classifying Failure Situations

At most times, failures in MAS/MRS affect single agents possibly controlling robots that typically are no longer able to perform with the same quality as before. However, failures may not impair all functionality of the agent if only parts of the agent-controlled hardware suffer from the failure, i.e., if only some S&A are affected. Of course, failures affecting one agent alone may also impair other cooperating agents that rely on the failing agent. This may lead to the total failure of whole ensembles if the failing agent executes a task that is indispensable for achieving the collective's cooperative goal. Like in other approaches for controlling MAS/MRS in the current literature, e.g., the approaches described in [Cui et al., 2014; Coltin and Veloso, 2013; Faci et al., 2006; Koppensteiner et al., 2009], we must face this problem in Multipotent System for ensuring the user-intended handling of plans with robust ensemble programs. Depending on when and where a failure occurs in a Multipotent System's up-time, the impact to the system is different. In every case, the set of roles an agent can adopt (cf. Section 3.2) becomes restricted when the agent suffers from a failure. Thus, agents may not be able to adopt specific roles necessary for participating in future ensembles or cannot execute their current role in an ensemble any longer. To point out necessary future

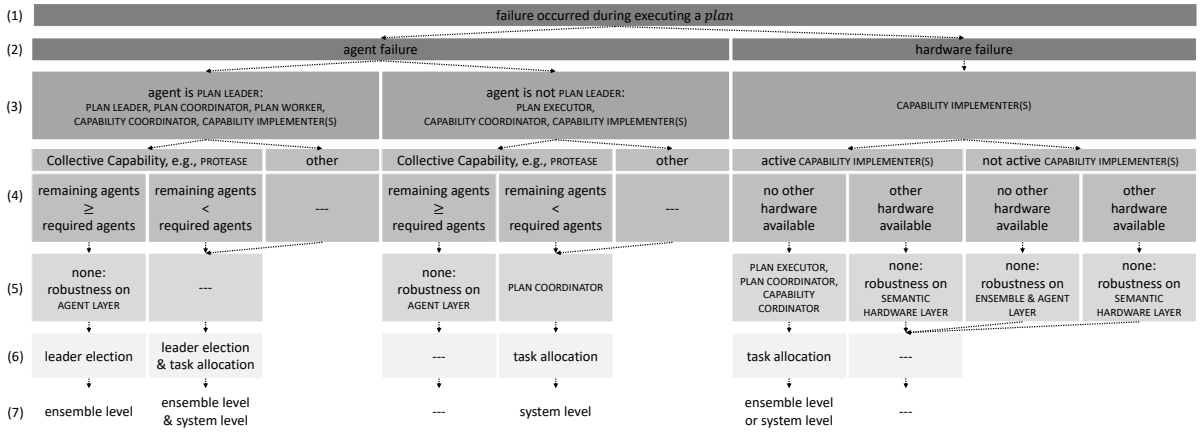


Figure 7.21: Impact of failures happening during executing a plan: (1) Failure situation, (2) failure level, (3) directly affected roles, (4) additional criteria influencing necessity of countermeasures, (5) indirectly affected roles, (6) countermeasures, (7) scope of countermeasure.

improvements, we briefly elaborate the influence of a failure during the execution of a plan. We can identify two situations I. and II. during the handling of a plan each being differently affected by agent or hardware failures.

When a failure happens before the user introduces the plan to the system (plan preparation in situation I), our self-aware, market-based mechanism for Ensemble Formation (cf. Chapter 5) inherently compensates for the failure: If the failure affects the hardware connected to an agent, the core functionality of that agent on all layers of the Multipotent Systems reference architecture is not influenced. Like after an intentional physical reconfiguration of the agent's hardware (cf. Chapter 6), the agent may lose one or more of its provided capabilities that depend on the failing hardware (cf. Section 3.2.8) but does not lose any of its other functionality. However, if the failure affects the whole agent, it can no longer adopt any role and can not execute any functionality anymore, including the communication with other agents. In situations of type I, this is only severe if there is no other set of agents in any hardware configuration that can handle the plan. In that case, the Multipotent System has depleted all of its redundancy and self-adaptation possibilities, and thus measures outside the system are required (e.g., the user needs to refine the plan). Consequently, we do need not require any other technique for detecting and compensating for failures in situations of type I as the self-organization mechanisms for Ensemble Formation and Physical Reconfiguration already included in our approach are sufficient for this.

More critical are situations when a failure happens during the execution of a plan (situation II). When analyzing the state of the agents' role adoption in an ensemble during the execution of an ensemble program for an exemplary plan ρ, the classification of the situation is nontrivial. Therefore, we investigate in this in more detail in Figure 7.21. Like in situations of type I, it is again relevant whether only connected hardware or a whole agent fails. Furthermore, also the roles the failing agent adopts for handling the plan have a different impact and require different countermeasures on different scopes (cf. Figure 3.3). During the execution of an ensemble program, an agent can either be a PLAN WORKER only executing an EPU program or adopt further roles, e.g., that of the PLAN LEADER and PLAN COORDINATOR. This leads to different cases of possible failures, each having different degrees of severity. A *hardware failure* influences the

execution only if one CAPABILITY IMPLEMENTER exclusively depends on this concrete hardware and the CAPABILITY IMPLEMENTER is also actively used in the plan (additional criteria in Figure 7.21). Then, the affected agent loses its role of a CAPABILITY IMPLEMENTER and thus can no longer appropriately adopt its role of a PLAN WORKER. Otherwise, the agent can self-heal from the failure within its role of a SELF-AWARENESS PROVIDER that provides robustness in such situations on SEMANTIC HARDWARE LAYER. The SELF-AWARENESS PROVIDER can detect the failure of the hardware and reassign respectively redundant hardware to the CAPABILITY IMPLEMENTER if available. In any case where the affected capability implementer is not actively used in a plan, the failure has no impact on the plan's execution and thus can be handled as described for situation I. If the affected agent no longer can adopt its role of a PLAN WORKER, this typically also has consequences for the agent adopting the role of a PLAN COORDINATOR because the ensemble no longer can work as intended. If so, we require an appropriate countermeasure to reestablish a working state in the system and compensate for the now missing PLAN WORKER. Fortunately, we can make use of the already established mechanism for Ensemble Formation (cf. Chapter 5) and *Physical Reconfiguration* (cf. Chapter 6). We first can apply them on the scope of the current ensemble to find a valid allocation in a minimal-invasive process. If this is not possible, we can include also other agents in the mechanism. If there is any solution to the respective problem of Ensemble Formation, we find it that way.

The same situation occurs when instead of a *hardware failure* an *agent failure* occurs for an agent that is *not* the PLAN LEADER. The consequently failing role of a PLAN WORKER the failing agent adopts also affects the PLAN COORDINATOR. Again, we can exploit our mechanisms from Chapters 5 and 6 as a countermeasure to self-heal from that situation. This time, we need to perform this on the scope of the whole Multipotent System because we already know that there are not enough agents within the ensemble anymore (cf. Figure 7.21).

The only situation where the failure of a PLAN WORKER has no further consequence for the PLAN COORDINATOR is that when the PLAN WORKER was a participator in a Collective Capability, e.g., executing PROTEASE. If enough redundancy is present in the collective, the collective can inherently compensate for the failures occurring at individual agents. Thus, up to a certain amount of failures, i.e., if the condition holds that in the collective the *remaining agents* $\geq$ *required agents*, the ensemble can compensate in a self-organized fashion. If this redundancy is depleted, i.e., the condition holds that *remaining agents* $<$ *required agents*, we need another countermeasure to self-heal from that situation. Again, we can refer to the mechanisms for Ensemble Formation and Physical Reconfiguration from Chapters 5 and 6. Things can get even worse if the agent failure affects not only a PLAN WORKER but the PLAN COORDINATOR. In this situation, the system requires to reassign the failing roles with additional countermeasures, i.e., elect a new PLAN LEADER with leader election for finding a new PLAN COORDINATOR. This search can take place within the ensemble suffering from the failure first, before reassigning the missing other roles with task allocation on the scope of the Multipotent System. Again, the severity of the situation can be reduced if the ensemble was executing a Collective Capability before the failure happened. In that situation the system only needs to reassign the task leader role with *leader election* on the scope of the ensemble because remaining agents can compensate for the failing role of a PLAN WORKER on AGENT LAYER.

7.7.1.2 Handling Failures with an Observer/Controller Mechanism

We propose a concept for dealing with failures occurring during task execution on the principles of a bottom-up failure detection and diagnoses mechanism combined with a top-down application of countermeasures. Therefore, we can extend our architecture from Figure 3.3 with new roles for each agent to realize an adapted version of the distributed observer/controller (O/C) pattern Schmeck et al. [2011].

Because failures can occur on different layers of our Multipotent Systems reference architecture, we also require integrating appropriate observing instances on these different layers and their adjacent layers. To enable the system to autonomously perform a failure diagnosis precisely, these observers frequently require up-to-date information about the instance they observe (i.e., the role). To achieve this on all layers, we introduce a healthiness mechanism similar to that of Koppensteiner et al. [2009], where lower-level instances send messages to higher-level instances containing information on the sender's state. If healthiness messages do not arrive within a respective timeout, the observing instance knows the observed instance suffered a failure. Thereby we realize the *observe* part of the O/C pattern, enable the system to detect the failure level and analyze the situation concerning the failing roles, additional criteria, and whether additional roles are affected according to the content of healthiness messages (cf. Figure 7.21).

Depending on the failure and where it gets detected by observers in the Multipotent System, the role affected by the failure can decide on the appropriate countermeasure and scope (cf. Figure 7.21). We differentiate between *reflex* and *deliberate* counter measures. With reflexes defined for every role, we aim to stabilize the current failure situation and avoid further failures by aborting any task-related execution in the ensemble. With deliberation (i.e., market-based task allocation and leader election), we compensate for the failure and bring the system back to an operating state.

Failures Concerning Hardware: To detect failures on the SEMANTIC HARDWARE LAYER, each agent observes the state of each piece of connected hardware by adopting a dedicated HARDWARE OBSERVER role. In case of a failure, the hardware observer informs the SELF-AWARENESS PROVIDER. The resulting control action only modifies the available capability executors if there is no other hardware to compensate for the failure.

Failures Happening on SEMANTIC HARDWARE LAYER: For each adopted role of a CAPABILITY IMPLEMENTER, the agent observes its internal state with a dedicated CAPABILITY OBSERVER on SEMANTIC HARDWARE LAYER. In case of a failure, the CAPABILITY OBSERVER informs the internal CAPABILITY COORDINATOR on AGENT LAYER which in turn escalates the information via the PLAN WORKER up to the respective PLAN COORDINATOR. The PLAN COORDINATOR then decides to involve the PLAN AUCTIONEER if necessary. The PLAN AUCTIONEER can initiate a new task allocation within the ensemble or the whole system depending on the current situation (cf. Figure 7.21).

Failures on AGENT LAYER: The agent adopting the PLAN COORDINATOR role observes each PLAN WORKER in the ensemble with a respective new role of an AGENT OBSERVER. If one PLAN WORKER fails, the agent observer informs the PLAN COORDINATOR which again decides on the necessity of a control action from the PLAN AUCTIONEER (task allocation with the scope appropriate for the situation, cf. Figure 7.21).

Failures on ENSEMBLE LAYER: All agents in the ensemble observe the PLAN COORDINATOR with a respective ENSEMBLE OBSERVER. If the PLAN COORDINATOR fails, the ENSEMBLE OBSERVERs inform the local LEADER VOTERS. These cooperatively perform a control action to

elect a new leader in the ensemble. The agent that then adopts the role of the PLAN LEADER decides on whether additional control actions are necessary because of additional failures, e.g. if task allocation is required.

Failures on PLAN LAYER: Because PLAN COORDINATORS frequently inform the local PLAN DISTRIBUTOR on the progress of a plan's execution, all agents can observe if that plan's execution has stopped by adopting an additional role of a PLAN OBSERVER. If so, they can collectively perform a control action to reset the current plan's execution status and continue as if the plan was newly introduced to the system. Because in some cases this *restart* of a plan is unfavorable, also the user device can act as a singleton PLAN OBSERVER to inform the user who then can decide on an appropriate control action, e.g., by redefining the plan.

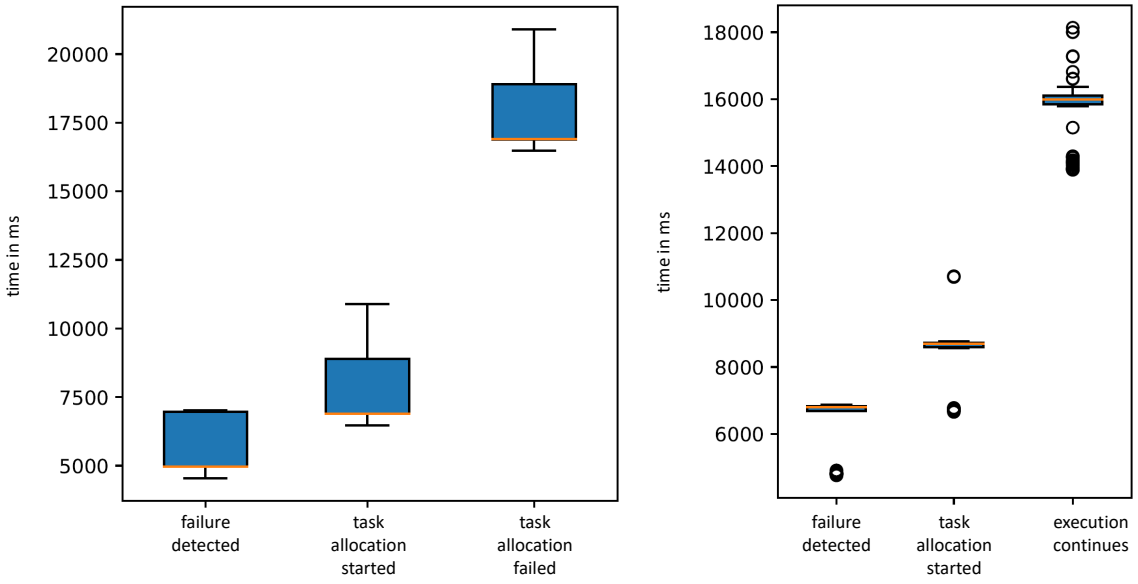
7.7.1.3 Preliminary Results

Evaluating the feasibility of integrating our concept in Multipotent Systems already resulted in some preliminary results. These results indicate that failure detection and recovery as we describe works generally and takes an adequate time in a prototypical implementation that is mainly influenced by the frequency we define for healthiness messages. Thus, improving the robustness of plan execution does not increase the efficiency with additional calculation times.

We tested the concept within different scenarios. In the 1st scenario, we evaluated the time required to detect a situation requiring external guidance as one urgently necessary hardware module fails while working on a plan. Figure 7.22a shows the result of 100 runs where the goal was to detect a tester-induced failure and recognize that the situation can not be handled autonomously by the ensemble. We see that with our healthiness messages-based concept, we can detect failures rapidly after their induction. On SEMANTIC HARDWARE LAYER, the detection happened after 5.69 seconds on average (avg) with a standard deviation (std) of 0.97 seconds. On ENSEMBLE LAYER, the failure was detected after 7.77 seconds avg (1.24 std). That external help is required was detected on PLAN LAYER after 17.78 seconds avg (1.24 std). Average times we receive for this scenario are mainly driven by the pre-configured frequency of sending healthiness messages. Default timeouts are 5 seconds for the HARDWARE OBSERVER on SEMANTIC HARDWARE LAYER, 2 additional seconds for the CAPABILITY OBSERVER on AGENT LAYER that informs the PLAN COORDINATOR. The PLAN COORDINATOR again takes additional 10 seconds, which is the default timeout we wait for proposals after triggering a task allocation as countermeasure.

In a 2nd scenario, we created a situation where compensation for the failure is possible within the ensemble by reallocating tasks, e.g., two agents can switch their tasks after one agent can not handle its current task anymore due to hardware failure. Executing 100 runs for scenario 2 resulted in a 100% success rate for compensating for the failure with the proposed measures (which we expect as we do not assume communication errors to happen, and thus the existing solution is also found during task allocation). We plot the results concerning the required time for detecting the failure and successfully compensating for it in Figure 7.22b. Again, we see that the failure was detected in a similar amount of time compared to the 1st scenario after 6.33 seconds avg (0.85 std). After additional 7.07 seconds avg (0.02 std) also the PLAN COORDINATOR gets aware of the situation and starts a task allocation with a CfP (cf. Chapter 5). Finding a new allocation within the ensemble and continuing the plan with this new task assignment then takes 15.69 seconds avg (0.94 std).

The preliminary results we found in these two experiments point towards the feasibility of applying countermeasures as we propose them in our O/C pattern adaptation within an



(a) 1st scenario (b) 2nd scenario

Figure 7.22: Box-plots for 100 runs of both scenarios, results from [Menssen, 2019].

appropriate time frame, if not happening to frequently. Future evaluations of our concept can give final proof that its integration into Multipotent Systems can increase robustness without causing too much influence to the run time to be deployed to flying ensembles.

Chapter Summary and Outlook

In this chapter, we proposed our approach of executing ensemble programs we can program with our approach MAPLE in Chapter 4. Our findings within Chapters 5 and 6 allow us to assume that we have appropriately configured ensembles at hand for doing so. Our approach for the distributed and synchronized execution of ensemble programs can handle the control structures, ensuring the correct program control flow of ensemble programs we generate with MAPLE. Combined with the correct handling of the data flow within ensemble programs, our approach supports sequential, parallel, conditional, and repeated control flow. Furthermore, it supports the run-time generation of new ensemble programs using replanning. In addition to coordinating the execution of physical and virtual capabilities, our approach also allows for the execution of Collective Capabilities that involve not only particular agents but also whole collectives, e.g., for achieving emergent effects using swarm behavior. We illustrate how we can realize such Collective Capabilities within our approach with two parametrizable Collective Capabilities. In a first Collective Capability c^v_{PROTEASE}, we encapsulate our finding of an Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE) that allows for the generic implementation of swarm behavior. Using different sets of parameters for PROTEASE allows for changing the specific swarm behavior within an ensemble executing it. Compared to other approaches for realizing swarm behavior in current literature, PROTEASE thus reduces the required engineering effort. In a second Collective Capability c^v_{EXT}, we encapsulate an abstract adapter for executing programs dedicated to controlling aggregates/swarms/ensembles generated with other approaches for ensemble programming. In our evaluations, we first demonstrated the generality of PROTEASE for producing different swarm behavior by only changing its parameters. We further illustrate how different instantiations of PROTEASE can beneficially be used in different case studies and how different sets of parameters producing the same emergent effect compare to each other. Second, we evaluate our execution engine for ensemble programs within our reference implementation of Multipotent System, which we introduced in Chapter 3. We do this by providing video materials showing the execution of the examples we introduced in Chapter 4 that cover all possible control and data flow structures we allow in MAPLE. Furthermore, we evaluate our integration of Collective Capabilities into ensemble programs. We demonstrate how a c^v_{PROTEASE} enables the goal-oriented exploitation of different emergent effects we can produce using PROTEASE. Moreover, we demonstrate how we can instantiate c^v_{EXT} with the approach of aggregate programming Protelis, enabling us to execute Protelis programs in a goal-oriented fashion in Multipotent System. Furthermore, we give preliminary results we achieved for increasing the robustness of plan executions using an adapted Observer/Controller concept for detecting agent and hardware failures.

Chapter 8

Achievements of this Thesis

This thesis was motivated by the potential that mobile Multi-Robot Systems (MRS) and especially ensembles of flying robots like Unmanned Aerial Vehicles (UAV)) provide for a wide range of different applications and use cases. We aimed at easing access to this potential by developing an approach for Mission Programming for Flying Ensembles. By enriching the measures of classical autonomous planning with mechanisms implementing the paradigm of self-organization, we combine the benefits of both research areas. To enable this combination, we provide a Multi-Agent Systems (MAS) reference architecture for flying ensembles allowing for the runtime adaptation of the physical configuration of robots controlled by respective software agents. In this chapter, we summarize the content we presented throughout the last chapters for achieving this goal in Section 8.1. We then elaborate on possible future work that this thesis enables in Section 8.2. Finally, we conclude this thesis by outlining its contributions to the research community in Section 8.3.

8.1 Summary of this Thesis

In the first Chapter 1, we introduced the problem of Mission Programming for Flying Ensembles and proposed our general idea for solving it with our approach of Combining Planning with Self-Organization. Because of the diversity in different fields of research, the mechanisms we adapt and integrate stem from, we used this introduction to clarify the sometimes differently used terminology. For doing so, we analyzed the current developments, and related work others propose for commanding MAS/MRS in versatile use cases and environments. Moreover, we used this analysis to sharpen our focus on flying ensembles involving multi-rotor controlled UAV. In our literature research, we emphasized approaches we can beneficially apply for such flying ensembles. Summarizing on the current state of the art of current approaches for Mission Programming for Flying Ensembles, we detected two key drawbacks we aim to overcome with our approach: First, because of their high software-specific application orientation or their hardware-specific implementation, achievements of many approaches can only hardly be transferred to other use cases than those they were initially designed for. Second, most current approaches can only efficiently handle complexity when this complexity is introduced by either the mission design or uncertain environments. Current approaches handle complexity in mission design typically by designing agents heterogeneously concerning their capabilities and providing classical autonomous planning measures for instructing the agents. For handling uncertainty introduced by the environment, current approaches often rely on the paradigm

of self-organization instead. For handling uncertainties, self-organization provides robustness against failures of individuals by redundancy in homogeneous systems and offers measures to handle large-scale systems by making most decisions locally. To overcome these two drawbacks, we proposed integrating both paradigms exploiting their benefits in an approach for Mission Programming for Flying Ensembles by Combining Planning with Self-Organization.

During our initial literature overview in Chapter 1, we furthermore detected the application class of Search, Continuously Observe, and React (SCORe) missions we aim at with our approach for Mission Programming for Flying Ensembles. With their different parts, SCORe missions subsume many tasks from applications in current Search and Rescue (SAR) case studies, Environmental Monitoring case studies, Distributed Surveillance case studies, and Major Catastrophe Handling case studies we can find in the current literature. In Chapter 2, we then introduced four different case studies that involve partial or complete SCORe missions, which we used throughout this thesis for evaluating and demonstrating the achievements of our approach.

In Chapter 3 we introduced the base we rely on for realizing our approach for Mission Programming for Flying Ensembles: The Multipotent Systems reference architecture we use to integrate the necessary concepts and algorithms for Combining Planning with Self-Organization. In our understanding, Multipotent Systems define a new system characteristic for MAS/MRS besides the currently existing characteristics that define each system to be either homogeneous or heterogeneous concerning its agents'/robots' capabilities. With Multipotent Systems, we aim at overcoming the first drawback we named above. Because in Multipotent Systems, we designed the connection between agents and their capabilities to be flexibly adaptable at runtime, systems implementing the concepts from our reference architecture can be applied not only to one specific but to many different use cases. Depending on how the requirements of a specific situation are, agents in Multipotent System can become heterogeneous specialists or become homogeneously configured swarm-like entities. By providing inherent self-awareness features combined with appropriate software and hardware design, we enable robots controlled by agents that are designed following our reference architecture to be physically reconfigurable at runtime. Moreover, the design of Multipotent System enabled us to integrate the required measures for dealing with the second drawback of current approaches we can find in the literature on controlling MAS/MRS, i.e., combine methodologies of classical planning with the paradigm of self-organization. We described how we adapted existing technologies and integrated them with newly designed ones for achieving that goal on the different layers of our reference architecture. We validated this integration by successfully deploying and using a prototypical implementation of our reference architecture to real hardware that made use of the included algorithms and technologies in multiple sets of laboratory and field experiments.

To instruct systems that implement the concepts of our reference architecture for Multipotent System, we propose a Multi-Agent Script Programming Language for Ensembles (Maple) in Chapter 4. To compete with other programming languages from the literature and allow for the goal-oriented instruction of homogeneously and heterogeneously configured system, we integrated the necessary features from both sides in Maple. In Maple, we can design sequential, parallel, conditional, repeated, and concurrent program control flow with the approach of Hierarchical Task Networks (HTN) we adapt accordingly. That way, we can let the executing system generate respective ensemble programs situation-specific by using automated planning. Maple thus enables the design of complex SCORe missions using program control flow structuring measures known from task-orchestration approaches to instructing either individual agents or whole groups of agents. By abstracting concrete agents and the robots they control

in MAPLE, we introduce an additional degree of freedom. The system that executes programs generated with MAPLE then can exploit this freedom the ensemble programmer allowed for if the individual situation requires it. Using agent groups instead of concrete agents when designing ensemble programs, the ensemble programmer can decide who executes instructions defined in the ensemble programs to the executing system itself. Combining the possibility to command the execution of swarm behavior and quantifying its respective emergent effect besides commanding the execution of other capabilities, MAPLE provides all measures to instruct both homogeneous and heterogeneous systems using the combination of planning and self-organization. Our evaluation of MAPLE demonstrates its flexibility and expressiveness by example and by comparison with other approaches for task-orchestration and ensemble/aggregate/swarm programming. With MAPLE, we thus solve the problem of *Task Scheduling* for Multipotent Systems by enabling an ensemble programmer to define *what* needs to be performed by an ensemble *in which order* of execution.

For deciding on *who* should execute the instructions scheduled in an ensemble program and making use of the flexibility an ensemble program designed with MAPLE allows for, we proposed a mechanism for the self-organized Ensemble Formation in Chapter 5 which we identified to be an instance of the *Task Allocation* Problem. Our mechanism takes care of forming ensembles that can fulfill all requirements of any ensemble program we can define with MAPLE. This includes selecting an appropriate subset of agents within the Multipotent System and taking care of the correct configuration of each agent concerning its capabilities. The measures we integrate into this mechanism use the self-awareness of each agent to take into account the flexibility in the agents' configurations that can change over time due to physical adaptations of their respective hardware. We achieve this by adapting the methodology of market-based task allocation that is well established in the field of MAS/MRS for our specific scenario. Besides dealing with the changing composition of agents that change their qualification for executing tasks defined in ensemble programs, the distributed nature of our market-based approach can deal with possibly high numbers of agents involved. We evaluated the feasibility of forming appropriate ensembles for ensemble programs generated with MAPLE with different theoretical and practical experiments. Those evaluations show that our approach for a distributed, self-aware market-based Ensemble Formation can handle high agent numbers if required in an ensemble program. Moreover, we could demonstrate that we can successfully apply it to real hardware by controlling a flying ensemble in indoor and outdoor experiments.

In Multipotent System, we allow for the physical reconfiguration of robots and thus can exploit the possibility of adapting the capabilities the agent controlling the robot can provide. This causes that the requirements defined by ensemble programs and the configuration of agents might not perfectly fit each other in any situation. We thus required to decide on the question of *in which configuration* agents within an ensemble should execute an ensemble program. We found that this problem is an instance of the Resource Allocation Problem (RAP). Thus, in Chapter 6 we proposed a mechanism for solving the RAP by calculating new physical configurations for the robots in a Multipotent System in a self-organized fashion. To deal with the complexity of the RAP when increasing the number of agents in a Multipotent System, the number of hardware devices we can use for physical configurations, the number of tasks defined by an ensemble program, or the number of capabilities that are addressed in an ensemble program, we proposed a solution for decomposing the RAP into sub-problems. We empirically evaluated that this decomposition combined with a subsequent aggregation of partial solutions in most cases achieves a quality of the overall solution comparable to that of a centralized and thus optimal approach but requires only a fraction of its calculation time.

We apply this finding to enable deploying our solution to real hardware we can use in flying ensembles. That way, we enable the key feature of Multipotent System, i.e., the situation-dependent and requirements-oriented adaptation of the system's agents' configurations.

By introducing new degrees of flexibility within Multipotent Systems, we allow an ensemble programmer to use during the creation of ensemble programs with MAPLE and that we exploit when forming appropriately configured ensembles, we achieved to have everything prepared for finally executing ensemble programs. In Chapter 7 we described how ensembles could cooperatively perform this execution. The execution of ensemble programs can involve complex program flow structures that integrate the execution of instructions performed by individual agents with such requiring multiple agents or even swarms of agents to work together. To ensure the correct execution of ensemble programs, we introduced a concept for their self-organized, coordinated, and synchronized execution. With our approach, we can execute any ensemble program generated with MAPLE, involving the situation-specific generation of new ensemble programs using replanning. To exploit the benefits of self-organization during the execution of enabling programs, we proposed integrating the goal-oriented execution of swarm behavior into the programs' control flow in our solution. We could do this because we found an Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE). We can execute PROTEASE using different parameters to achieve different swarm behavior and generate different emergent effects in versatile use cases. We evaluate the generality of PROTEASE by instantiating it with a broad set of exemplary sets of parameters, propose how to quantify the respective emergent effect, and use it within complex ensemble programs. To evaluate the feasibility of executing ensemble programs involving swarm behavior, we integrate our solution into our prototypical reference architecture. We demonstrated in a simulation environment how ensemble programs involving sequential, parallel, conditional, and repeated execution could also execute goal-oriented swarm behavior. In addition, we evaluate the possibility of integrating other approaches for commanding collectives into ensemble programs by demonstrating its feasibility with the example of Protelis Pianini et al. [2015] we can use for aggregate programming.

8.2 Future Research Directions

This thesis investigated many relevant topics for realizing our approach for Mission Programming for Flying Ensembles by Combining Planning with Self-Organization. Unless we achieved significant improvements in the different areas we tackled during working on this thesis, there are still topics left for future research. This comes because we either abstracted from details requiring further investigation and improvements of our and other mechanisms, we detected higher levels of complexity in specific areas, or our approach first enables future research to build on its achievements.

While our user interfaces for designing ensemble programs using the concepts of HTN already eases the programming of flying ensembles, it can be further improved concerning its usability for domain-only experts. Currently, besides correct and goal-oriented programs, we can also design useless programs in MAPLE. This starts with the possibility of designing unintended infinity loops using the concept of repeated executions we introduced and ends with the possibility to address per se infeasible requirements in ensemble programs, e.g., by including two incompatible Operator in one Primitive-Node instructing the same agent to move to two different positions at the same time. While programming-affine users may easily avoid such failures during programming after some initial introduction to the concepts of

MAPLE and their respective functionality, we can not assume every domain-only expert like rescue forces or firefighters to be aware of the possible pitfalls without additional guidance during programming. Furthermore, designing ensemble programs with MAPLE still require formal notation to define conditional program control flow or describe situations relevant for generating correct ensemble programs during planning. Because with MAPLE, we aim to ease the access toward the beneficial usage of the different potentials flying ensembles can provide in many different scenarios, we see the need to improve the current way of programming for ensembles.

Concerning planning with MAPLE, i.e., generating new ensemble programs at runtime, we see potential in further increasing the flexibility we provide to the system when deciding on the most appropriate plan. Currently, we allow for only one plan to be applicable for a specific situation. While we can include conditional program control flow into this plan, we restrict the self-organization possibilities for the executing system for the sake of better controllability, i.e., we currently emphasize the comprehensibility of the system's self-organized decisions. In principle, we can move even more decisions from the design-time towards the runtime by allowing for multiple possible solutions resulting from planning in specific situations. By enriching each of these solutions with relevant qualitative measures, e.g., the solution's efficiency, quality of the solution, resource demand of the solution, or the provided degree of robustness, we could allow the executing system to decide for the best solution regarding a user-defined optimization criteria. If, e.g., there are only a few agents available, the user requires a very precise result, the time required for generating the result is negligible, and the occurrence of uncertainties influencing the execution is very low, searching for an object of interest should be best executed by a single agent that can systematically explore the respective area of interest. If instead, there are enough resources available, the chance that uncertainties influence the execution is high, and the time to derive the solution is critical, searching for an object of interest should better be performed by a group of agents executing a respective swarm behavior. We think that adapting our current approach in MAPLE to support decisions as we describe them above, e.g., by extending our planning algorithm to a respectively adapted heuristic A* search that works with different quality measures the user defines for different solutions. Such improvements would even emphasize the benefits of combining planning with self-organization.

To allow for concurrent execution of plans and the ensemble programs we generate, we currently need to make some assumptions that we could overcome by integrating additional mechanisms and technologies into our approach. With MAPLE, we can generate concurrency explicitly by using Split-Nodes during the definition of a HTN and implicitly by using Replanning-Nodes embedded in the execution of an ensemble program. Because this concurrency discloses the problem of concurrent data modifications performed on variables in the world state when using Runtime-Worldstate-Modification-Nodes or Planning-Time-Worldstate-Modification-Nodes in more than one ensemble program, we currently need to ensure only one concurrent execution performs such data access. Because the literature on concurrent data access is vast and we already achieved the first results using a distributed and synchronized database, we think that appropriate extensions of MAPLE concerning data storage and management can also enable unrestricted concurrent executions.

During our evaluations we performed for the mechanism we propose for the self-organized physical reconfiguration of agents, we detected the high complexity of the problem. Increasing the problem size to a too high level for different parameters simultaneously (e.g., number of agents, number of hardware modules, number of tasks, number of possible capabilities)

leads to calculation times unusable when addressing flying ensembles. This comes because the approach we provide aims at finding close to optimal solutions regarding the number of required configuration steps, i.e., plug-in and plug-off steps necessary for achieving the new configuration. By softening this optimization goal and also accepting solutions providing reduced quality only, we could apply other measures for solving the RAP we need to solve. One option is that of using a greedy algorithm instead of a CSOP-based solution. Such a mechanism could substantially increase calculation times on the cost of dropping the quality of solutions. Currently, we develop such an approach. Thereby, we aim at solving the problem efficiently while still trying to keep the number of necessary plug-in and plug-off steps reasonable. We expect our greedy-algorithm solution to keep calculation times in a comparably low range to improve its applicability for even larger flying ensembles.

We see another possibility to improve on our current work with future research by increasing the robustness. We already introduced robustness to the execution of ensemble programs by integrating measures of self-organization, especially with our achievement of integrating strictly planned agent behavior with swarm behavior. If agent failures happen during the execution of PROTEASE (which we use for integrating swarm behavior into Multipotent Systems), the collective can compensate for that failure by measures of self-organization. If failures happen during the execution of other parts in an ensemble program, we require to integrate other measures for compensating them. In Section 7.7, we classified possible failures and proposed possible countermeasures for different kinds of failures, and outline a sketch for a possible mechanism for applying those measures for Multipotent System. While we already achieved some preliminary results on the general feasibility of the concepts we propose, we do not yet have a proof for it and require further studies to achieve that. Nevertheless, improving on the measures to handle uncertainty in real world systems by increasing the robustness of Multipotent System is an urgent step to perform for moving towards their actual application within many use cases.

In Section 3.4, we evaluated the feasibility of applying our approach of Multipotent System to real hardware and gave proof of concepts for instructing them using our approach for Mission Programming for Flying Ensembles. While we demonstrated that the same prototypical reference implementation could also execute any ensemble program we can generate with MAPLE, we could not perform the necessary experiments also using real hardware. Future research on this topic should focus especially on the execution of PROTEASE with such real hardware, including its current technical implementation. While our reference implementation for Multipotent Systems we use for our evaluations in simulation is very close to that we use for controlling real hardware-driven systems (we need to exchange the respective simulated hardware drivers to those of real hardware), we learned during our outdoor and laboratory experiments we describe in Sections 3.4 and 5.5.2 that many problems to be solved first arise on the way for closing the reality gap.

Another idea for future research we enable with this thesis is to investigate our approach's generality for flying ensembles. While most mechanisms we integrate should also work for other goal systems involving other hardware, e.g., task orchestration, task allocation, and resource allocation should also work appropriately when using driving, swimming, or diving robots, we might need to revise the mechanisms we provide for executing ensemble programs. The most relevant aspect to further investigate, in our opinion, would be the execution of swarm behavior using PROTEASE. Transferring the emergent effects and the possibilities for quantifying them as we introduced them in Chapter 7 to other systems, e.g., such that are restricted in their movement possibilities like driving systems, can result in deeper insights

concerning the generality of our approach PROTEASE.

In addition, we can think of many other possible applications of PROTEASE in other use cases. Making use of the scalability we achieve through swarm behavior can improve the performance and development of current applications involving flying ensembles. For light shows like that of Intel [2021], e.g., planning and pre-calculating the necessary trajectories for large scale ensembles currently is very time-intense. Further, the execution is controlled centrally and thus requires massive computational power compressed in one single device. Using PROTEASE, we can provide similar behavior while providing scalability and robustness by exploiting the benefits of swarm behavior. While current forms we can shape and fill using PROTEASE are not yet as complex as that demonstrated by Intel [2021], we are currently developing a solution that can fill and shape any convex body and combinations of such.

8.3 Conclusion and Results

In this thesis, we achieved to develop an approach for Mission Programming for Flying Ensembles by Combining Planning with Self-Organization. We reached that goal by integrating necessary mechanisms and technologies from different research fields with each other in a reference architecture for Multipotent Systems. By transferring the idea of multipotency from biology to technical systems, we introduced a new way of looking at Multi-Agent Systems (MAS) and Multi-Robot Systems (MRS). Like multipotent stem cells that serve as our source of inspiration, Multipotent Systems provide measures to adapt themselves for different functions in versatile applications and use cases. We achieve this flexibility by separating the concepts of software agents that control robots from the concept of capabilities the agents gain from different hardware configurations. This is an invention compared to other current research that typically assumes the named connection to be static.

During the conception and the development of our approach, we detected the class of Search, Continuously Observe, and React (SCORe) missions perfectly fitting to Multipotent Systems. Classifying possible tasks for a flying ensemble into the three named categories, SCORe missions subsume many case studies other research currently investigate separately. We hope that our classification of the different tasks and the elaboration on their interconnection can help other researchers focus on the necessary actions for making the transition from one SCORe part to another. From our point of view, sharpening the focus on that transition is urgently necessary to achieve real-world applicability for future MAS/MRS. Focusing on these transitions was the enabler for most inventions and adaptations of existing mechanisms we developed in the studies we presented in this thesis.

As complete SCORe missions require to control individual agents as well as whole groups of agents, we created a Multi-Agent Script Programming Language for Ensembles (MAPLE) enabling that. To the best of our knowledge, with MAPLE, we provide the first approach to integrate the goal-oriented execution of self-organized swarm behavior within strictly planned missions. Using MAPLE enables ensemble programmers to define the required framework for a mission, i.e., specify parts where precise control is necessary and allow for the executing system to act in a completely self-organized manner.

Because transitions from one task to another in SCORe missions can require completely different configured agents for executing them, we adapted existing mechanisms for task and resource allocation appropriately with the measures of self-organization and self-awareness that exploit the unique properties of Multipotent Systems. That way, we achieved realizing

the required measures for calculating and executing physical adaptations of agents in Multipotent Systems with our approach for a Task and Resource Allocation Strategy for Multi-Agent Systems (TRANSFORMAS) in addition to the necessary measures required for forming appropriately configured ensembles for any SCORe mission we can formulate with MAPLE with our approach of a Distributed, Self-Aware Market-Based Mechanism for Ensemble Formation (SELF-MADE).

Furthermore, we achieved an execution engine for SCORe missions that involves measures for controlling and coordinating self-organization, making it usable in a goal-oriented fashion. Besides others, we achieved this by finding an Algorithmic Pattern for Trajectory-Based Swarm Behavior (PROTEASE), whose execution we embedded into our approach. With PROTEASE, we can now achieve different swarm behavior, quantify the emergent effect of its execution, terminate on its execution when appropriate, and process its result in the SCORe mission we embed it in by only modifying the parameters we use for executing its concrete implementation.

While we focus on the mechanisms and technologies necessary to realize Multipotent Systems on the more abstract and higher level of our layered reference architecture, we also achieved deploying and executing most inventions with real hardware modules. This was only feasible by combining the inventions of this thesis with that of another accompanying doctoral thesis that focuses on the lower architecture levels and provides the necessary measures for realizing self-describing hardware modules our approach builds on. In its thesis *Semantic Plug and Play - Fähigkeitsbasierte Hardwareanbindung modularer Robotersysteme*[1], Constantin Wanninger focuses on the required technologies for enabling self-description on the hardware level. The results of Constantin Wanninger will include innovative methodologies for the semantic annotation of self-descriptive hardware combined with reasoning mechanisms working on these descriptions. In addition to that, agents will be enabled to deduce the availability of capabilities provided by different hardware combinations and provide the information necessary for correctly executing these capabilities. By combing both research results, we thus can construct highly flexible and easy to instruct systems that integrate autonomous mechanisms enriched with the concepts of self-organization and enable their deployment to real hardware for actually applying them to real-world use cases in the future.

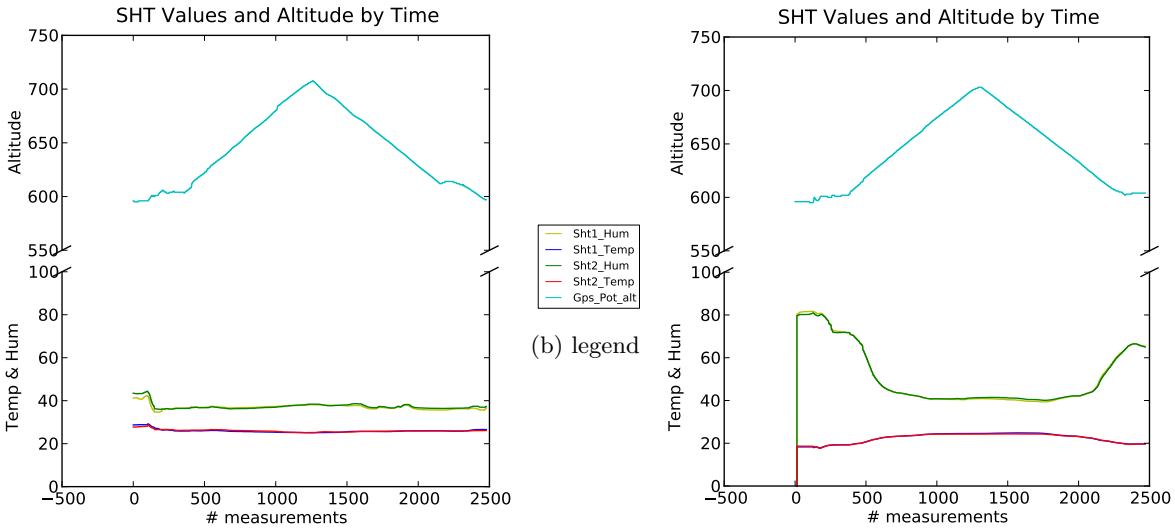
[1]German title, translated by the author: Semantic Plug & Play — Capability-Based Hardware Integration of Modular Robot Systems

Appendix A

Additional Evaluations For ScaleX 2015 and ScaleX 2016

A.1 ScaleX 2015

Figure A.1 supports our findings concerning the occurrence of an inversion and the effects of the phenomena. In there, we depict measurements derived by one single agent of our ensemble, derived at 13:00 local time derived at the southern measurement position (Figure A.1a) and 21:00 local time derived at the north-eastern position (Figure A.1c). In both figures, we see the

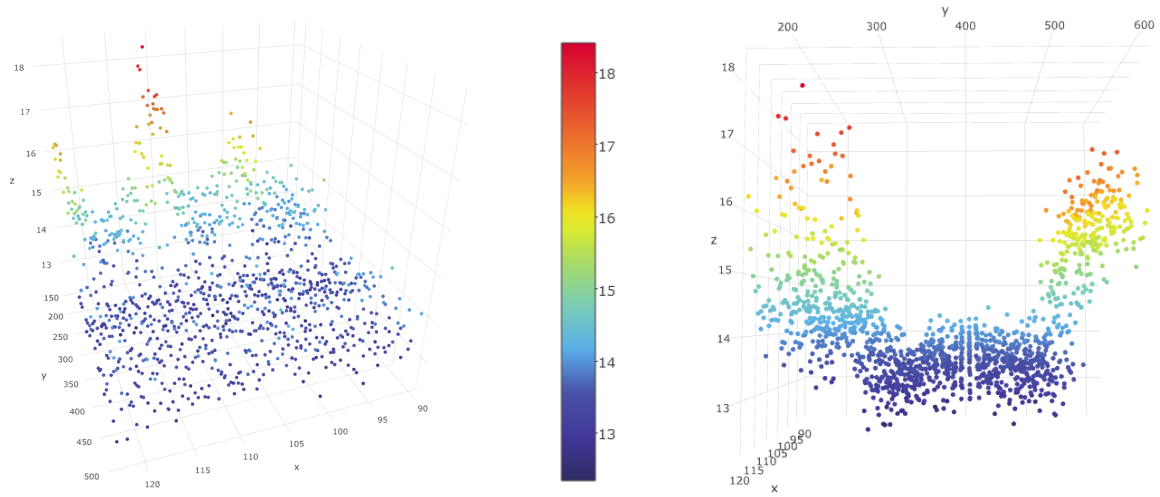


(a) Southern position, June, 30th at 16:00 o'clock local time.

(b) legend

(c) North-eastern position, June, 30th at 21:00 o'clock local time.

Figure A.1: Exemplary flight performed by one agent each, performed at different positions on June 30th, 2015. Compares the measured position with temperature and humidity measurements performed with the agent's SDH-prototypes encapsulating an Autoquad flight controller (Gps_Pot_alt on the top of the figures) [Autoquad, 2018] and two SHT-75 sensors (Sht_1 and Sht_2 on the bottom of the figures) Sensirion [2018].



(a) mobile DTS feasibility, top view (b) legend (c) DTS-measuements, side view

Figure A.2: Measurements performed by the mobile and airborne application of the DTS measuring technology, focusing on the measurements achieved with the fiber-optic cable that spanned between agents α_2, α_3, and α_4. We neglect measurements derived with the fiber-optic cable spanning between agent α_1 and α_2 because there, the fiber-optic cable was not positioned in the measurement height. Deep red color indicates the highest measured value. Deep blue color indicates the lowest measured temperature, temperatures in between are colored according to the color gradient in Figure A.2b. Figure A.2c shows the measurements depicted from a side view, with the starting position to the left and the landing position to the right. Measurement distribution over the z-axis of the plot depicts the cooling during the measurement flight. Figure A.2a shows the measurements from a top-down perspective, neglecting the post-plan execution phase.

position measurements of the agent performing the part of the measurement flight plotted in blue color to the top combined with the measured values for temperature and relative humidity for all sensors available in its configuration (i.e., we have data from both SHT-75 sensors encapsulated in the $\text{SDH}_{\text{SHT75}}$-prototype available). In Figure A.1a, we see that temperature and relative humidity do not vary much when changing the height at 13:00 local time. In Figure A.1c instead, we can see an increase in temperature and a drop in relative humidity coupled with increased height. Again, this indicates the occurrence of a temperature inversion, including its effect on relative humidity very clearly.

A.2 ScaleX 2016

In Figure A.2, we depict each of the measurements performed along the fiber-optic cable during another measurement flight in a 3-dimensional plot from different perspectives. Like in Figure 3.47, we depict measurements performed along the fiber-optic cable at x-y-positions indicating the measured temperature on the z-axis. Colors here depict the lowest measured values in deep blue, highest measurements in deep red, and temperatures in between according to the color gradient depicted in Figure A.2b. In the top-down view in Figure A.2a, again, we

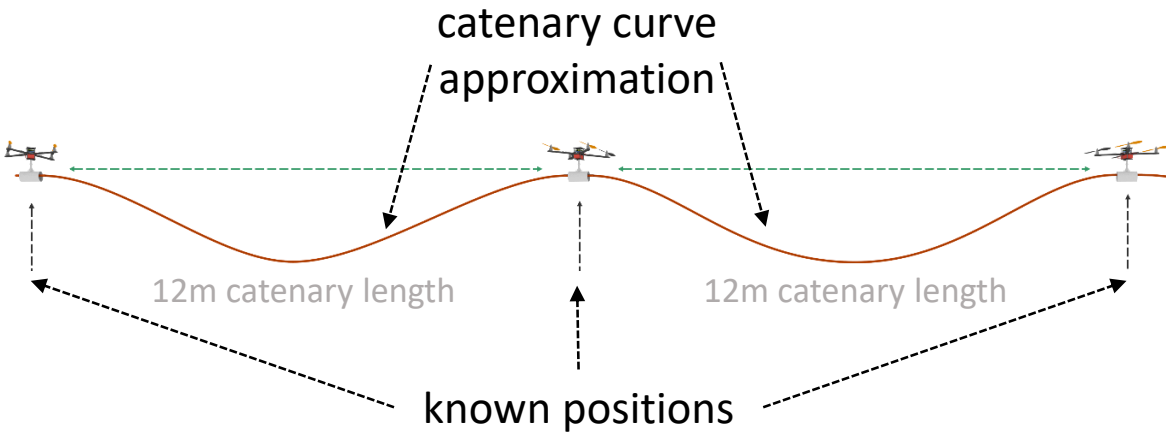


Figure A.3: Possibility of calculating the positions of measurements in-between agents that were carrying the fiber-optic cable more precisely. We can estimate the position of the cable despite its additional length between points we know. We know the positions of agents spanning the fiber-optic cable according to their position measurements performed with $c^{p}_{\text{M-POS}}$. Because we know the cable weight, length, and mounting points, we can calculate the catenary curve between two agents carrying the fiber-optic cable and thus estimated positions where individual measurements were performed while moving the fiber-optic cable through the air.

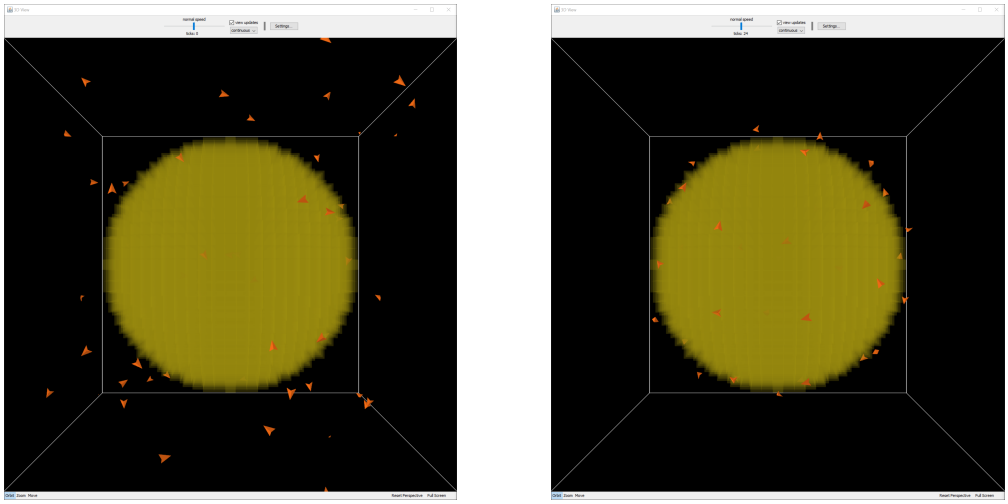
can see the decrease of temperature in higher heights when comparing measurements before and after takeoff. In this figure, we cut off measurements after finishing $\rho_{\text{SCALEX2016}}$. Also, in this measurement flight, we can see the *'temperature spikes'* before takeoff. This consolidates the impression that the phenomena of heating need to be taken into account when performing airborne measurements with flying ensembles, unless the fiber-optic cable acting as the sensor of the DTS does not show high inertia, i.e., cools down again very fast. The contrast gets even more emphasized when looking at the measurements from a side view in Figure A.2c. After measuring high values initially, the cooling down after starting the plan is rapidly visualized by the fast-changing colors of measurements. For further analysis, we can calculate the position of every measurement within Figure 3.47 and Figure A.2 even more precise, if necessary. We can calculate the centenary curve according to the distances between agents (i.e., $10m$ between agents $\alpha_{2,3,4}$ each) whose precise positions we know (from their execution of $c^{p}_{\text{M-POS}}$), combined with the additional fiber-optic cable length of $2m$ (cf. Figure A.3). This way, despite the safety measures we need to implement in the collective transport and its physical difficulties, we can determine measuring positions along the DTS even when performing measurements airborne and mobile.

Appendix B

Additional Content Concerning PROTEASE

B.1 Additional Evaluations for PROTEASE

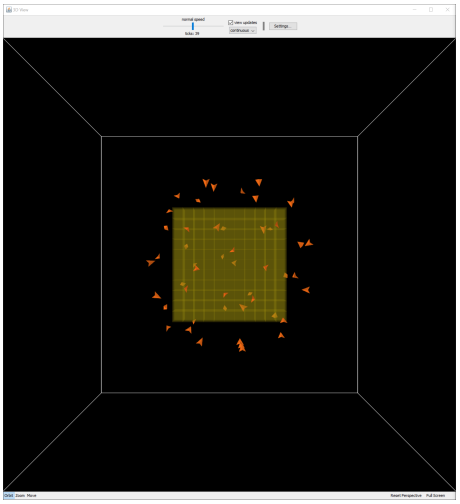
In Figures B.1 to B.4, we depict additional situations during the execution of Protease for realizing a Shape-Form swarm behavior. The image exports stem from our NetLogo implementation of Protease. While the agents in Figure B.1a are distributed randomlym, they shape the ball like form in Figure B.1b. When we change the form's shape to a cuboid in Figure B.2a, agents recognize they need to adapt their positions for reestablishing the desired shaping behavior. Figure B.2b shows the same situation from the side perspective, emphasizing the effect we produce by changing the form's shape. In Figure B.3a, agents now reshape the updated form. After changing the form's shape again in Figure B.3a, agents again get



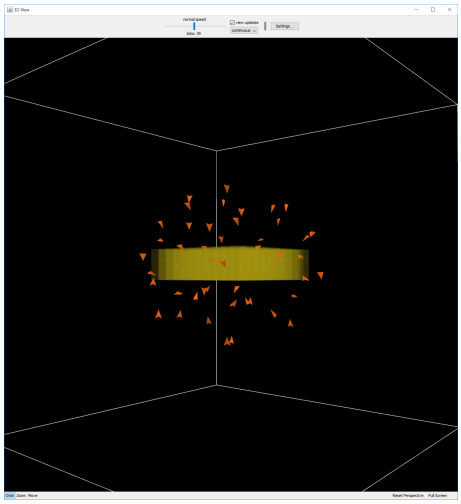
(a) randomized setup for shape behavior

(b) agents shape the ball-like form

Figure B.1: Image exports from the NetLogo simulation environment during executing Protease for a *Shape-Form swarm behavior*. The yellow-colored environment indicates the form to fill by the swarm.

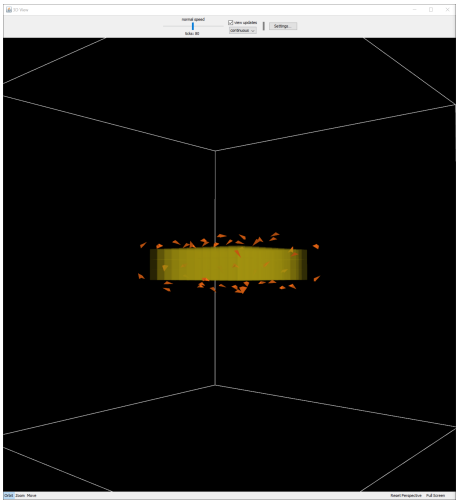


(a) changing the form to a cuboid

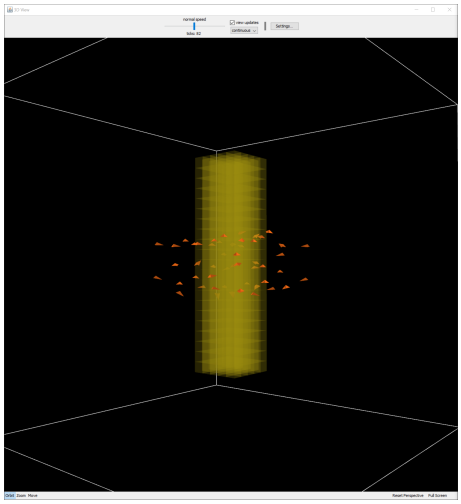


(b) side perspective on the cuboid

Figure B.2: Image exports from the NetLogo simulation environment during executing PROTEASE for a *Shape-Form swarm behavior*. The yellow-colored environment indicates the form to fill by the swarm.

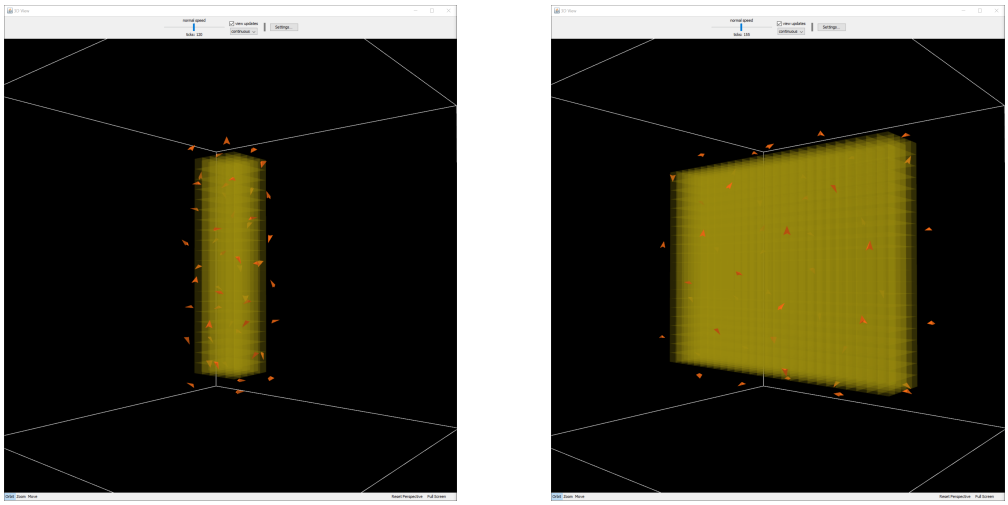


(a) agents fill the cuboid



(b) modifying the cuboid dimensions

Figure B.3: Image exports from the NetLogo simulation environment during executing PROTEASE for a *Shape-Form swarm behavior*. The yellow-colored environment indicates the form to fill by the swarm.



(a) agents fill the modified cuboid (b) agents fill a further modified cuboid

Figure B.4: Image exports from the NetLogo simulation environment during executing PROTEASE for a *Shape-Form swarm behavior*. The yellow-colored environment indicates the form to fill by the swarm.

aware they need to update their positions for the new shape. Figure B.4a then shows how agents also achieve shaping the form after the update. In Figure B.4b, we expanded the form again and depict the situation after all agents already adapted their positions for also shaping the expanded form.

B.2 Modeling Protease

Figure B.5 shows the class diagram focusing the relevant aspects of realizing the concepts of PROTEASE in our prototypical reference implementation of Multipotent Systems. We see the three of the four parameters defining PROTEASE, i.e., $\text{CALC}_{\text{PROTEASE}}$ in the form of the class *Calculator*, $\text{TERM}_{\text{PROTEASE}}$ in the form of the class *Terminator*, and $\text{AGG}_{\text{PROTEASE}}$ in the form of the class *Aggregator*. We do not require to express the concept of $\text{GROUP}_{\text{PROTEASE}}$ in this class diagram, as the respective group necessary for executing PROTEASE is formed dynamically using our self-organization mechanism for Ensemble Formation and Physical Reconfiguration we describe in Chapters 5 and 6. Together Calculator, Terminator, and Aggregator form the *SwarmAlgorithmDefinition*. The enumeration *SwarmAlgorihmDefinitions* is the adapter for integrating PROTEASE into our reference implementation of the MAPLE designer, i.e., realizes the possibility of planning the respective Collective Capability c^{v}_{PROTEASE} in HTN.

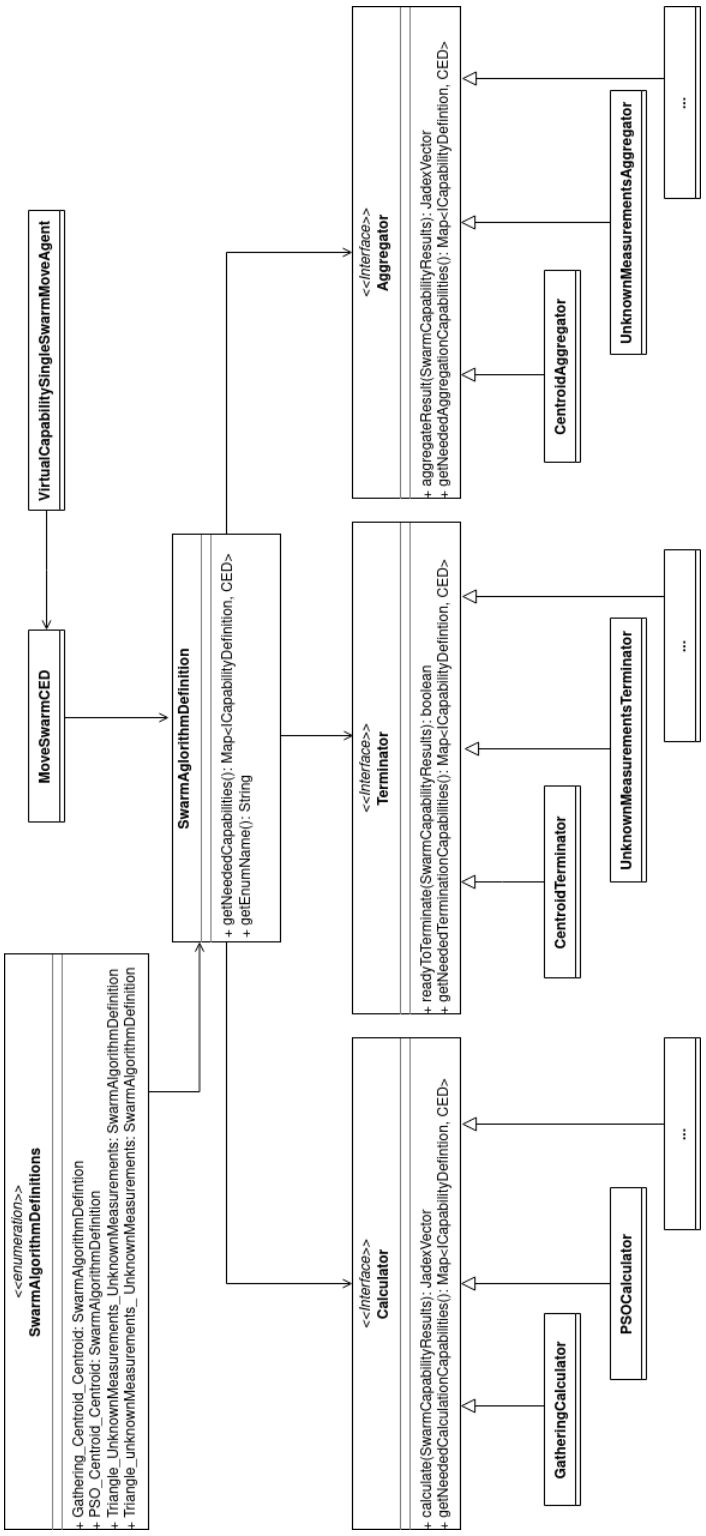


Figure B.5: Class diagram for the Collective Capability encapsulating the concepts of Protease.

Appendix C

Combining MAPLE Features

We present an exemplary HTN in Figure C.2 that combines many key elements of Maple with each other. Besides sequential, parallel (phyiscal and logical parallelism), conditional and repeated executions, it also involves Planning-Agents of different types, i.e., Identified-Planning-Agents, the $\alpha^{\rho}_{\exists}$, the $\alpha^{\rho}_{\forall}$, the $\alpha^{\rho}_{\{\text{MIN}\atop\text{MAX}\}}$, and the Set-Agent. We use goal-oriented execution of Collective Capabilities by integrating c^{v}_{PROTEASE} with different parameters. In the first plan (cf. Figures C.3a and C.3b) derived from the initial conditions in the world state, one Identified-Planning-Agent ${\alpha^{\rho}}_1$ step-wise increases its position concerning altitude and measures the gas concentration at its new position. As soon as the value exceeds a defined threshold, the agent moves pack to the position $\langle 0,0,0\rangle$ while another Identified-Planning-Agent ${\alpha^{\rho}}_2$ moves upward with a fixed velocity until ${\alpha^{\rho}}_1$ reaches its goal. Then also ${\alpha^{\rho}}_2$ finishes its execution (internal trigger in the ensemble). The ensemble then modifies a variable in the world state, gathers executing a respective Collective Capability, and subsequently triggeres a replanning. In the resulting plan (cf. Figure C.3c), another ensemble executes a Collective Capability encapsulating a *Ring-Of-Fliers* swarm behavior involving three swarm members using the Swarm-Agent $\alpha^{\rho}_{\{\text{MIN}\atop\text{MAX}\}}$ for a certain amount of time (100 internal steps), and then terminate the execution on its own. Subsequently, the ensemble modifies the variable relevant for replanning and starts a collective movement to the position $\langle 0,0,1\rangle$ using the All-Agent. In the last plan (cf. Figure C.1), a set consisting of Identified-Planning-Agents ${\alpha^{\rho}}_1$ and ${\alpha^{\rho}}_2$ moves to the coordinate $\langle 1,1,1\rangle$ in a first Primitive-Node before a not further specified agent $\alpha^{\rho}_{\exists}$ moves with a velocity until the capability gets terminated by the user while another Identified-Planning-Agent ${\alpha^{\rho}}_1$ moves with a velocity until canceling the execution is triggered internally, caused by the other agent stopping its execution, in a second Primitive-Node. We provide a video *Maple-Combined-Execution* showing the execution of the plans on GitHub and YouTube.

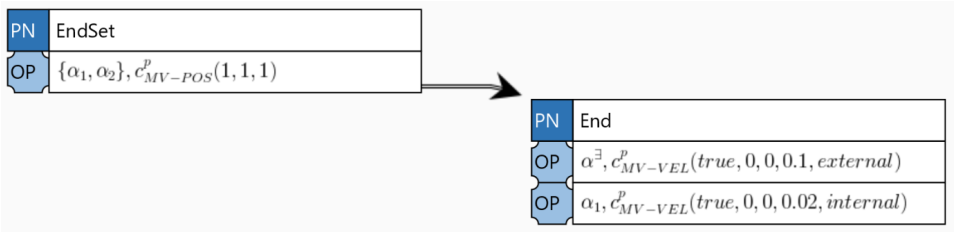


Figure C.1: Third plan created from the HTN in Figure C.2

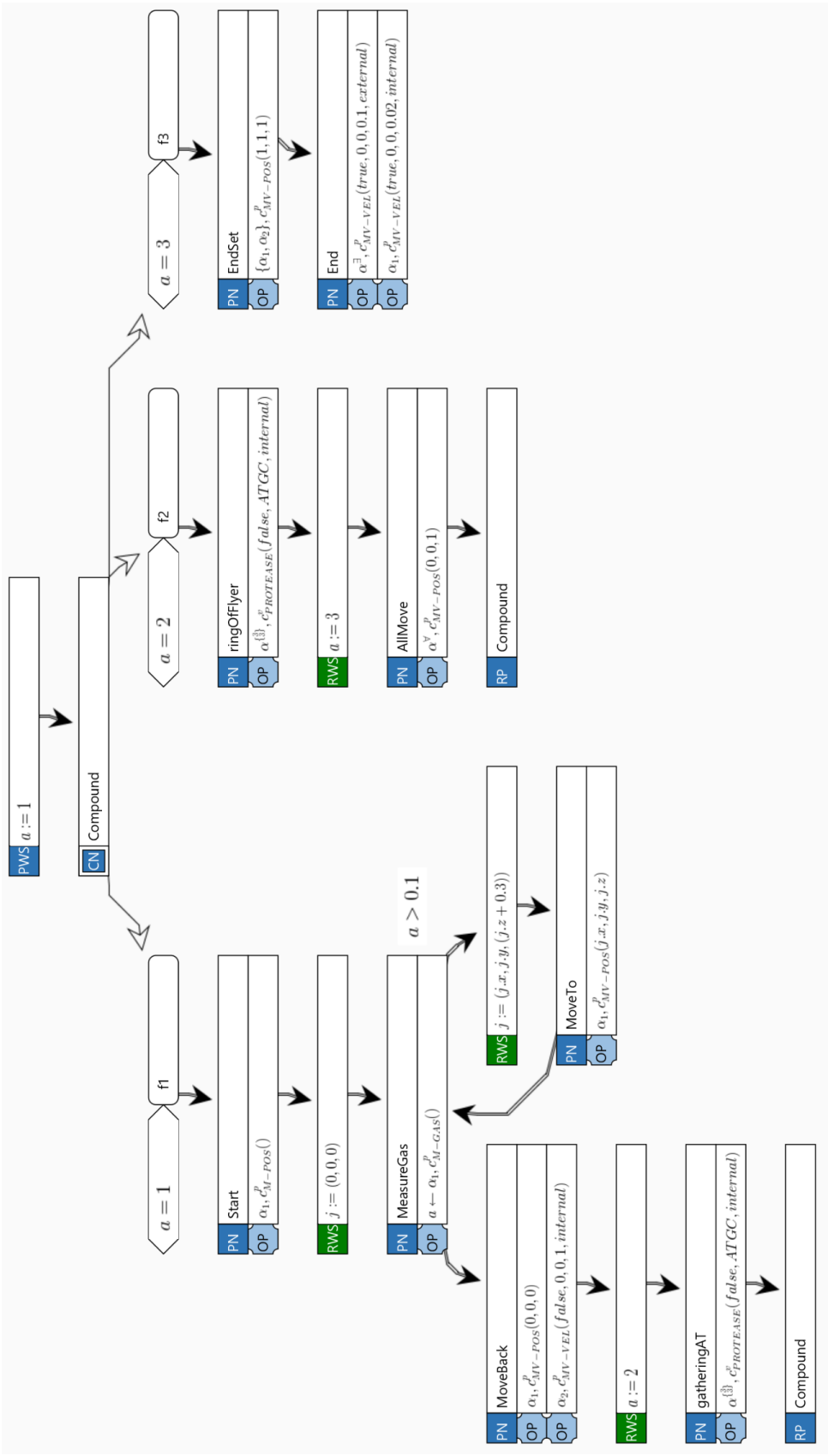
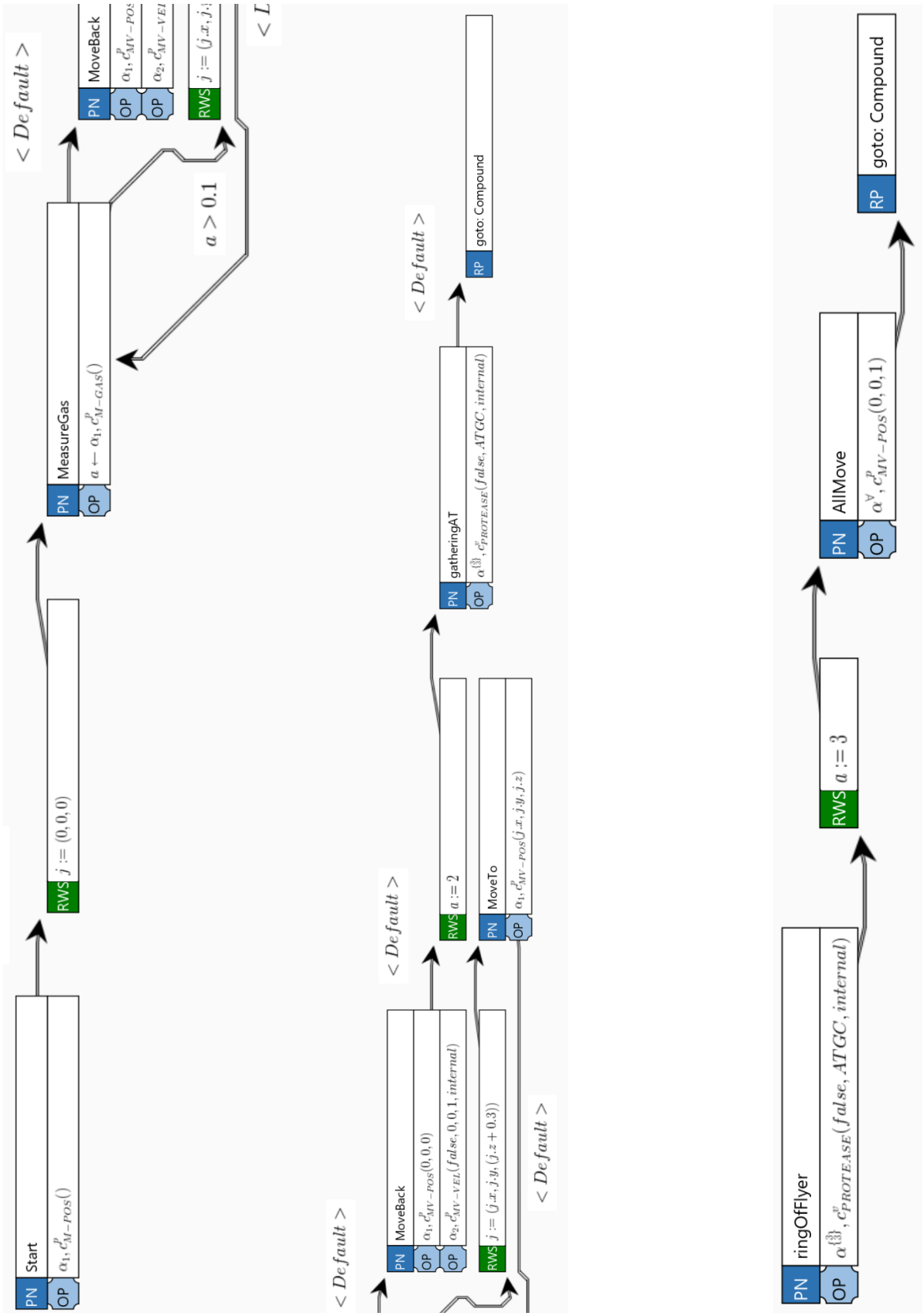


Figure C.2: HTN created with MAPLE integrating many of its possibilities.



(a) plan 1 - left part (b) plan 1 - right part (c) plan 2

Figure C.3: The initial plan created from the HTN in Figure C.2, figure split in two (left in Figure C.3a and right in Figure C.3b) for the sake of readability and the second plan in Figure C.3c.

List of Figures

List of Tables

List of Own Publications Cited in This Thesis

Anders, G., Siefert, F., Schiendorfer, A., Seebach, H., Steghöfer, J.-P., Eberhardinger, B., Kosak, O., and Reif, W. (2016). *Specification and Design of Trust-Based Open Self-Organising Systems*, pages 17–54. Springer Int. Publishing, Cham.

Hamann, H., Khaluf, Y., Botev, J., Divband Soorati, M., Ferrante, E., Kosak, O., Montanier, J.-M., Mostaghim, S., Redpath, R., Timmis, J., Veenstra, F., Wahby, M., and Zamuda, A. (2016). Hybrid societies: Challenges and perspectives in the design of collective behavior in self-organizing systems. *Frontiers in Robotics and AI*, 3:14.

Hanke, J., Kosak, O., Schiendorfer, A., and Reif, W. (2018). Self-organized resource allocation for reconfigurable robot ensembles. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 110–119.

Kosak, O. (2015). A decentralised swarm approach for mobile robot-systems. In *Organic Computing: Doctoral Dissertation Colloquium 2015*, volume 7, page 53. kassel university press GmbH.

Kosak, O. (2017). Facilitating planning by using self-organization. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 371–374.

Kosak, O. (2018). Multipotent systems: A new paradigm for multi-robot applications. In *Organic Computing: Doctoral Dissertation Colloquium 2018*, volume 10, page 53. kassel university press GmbH.

Kosak, O., Anders, G., Siefert, F., and Reif, W. (2015). An Approach to Robust Resource Allocation in Large-Scale Systems of Systems. In *Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th Int. Conf. on*, pages 1–10.

Kosak, O., Bohn, F., Eing, L., Rall, D., Wanninger, C., Hoffmann, A., and Reif, W. (2020a). Swarm and Collective Capabilities for Multipotent Robot Ensembles, currently under review. In *9th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation*.

Kosak, O., Bohn, F., Keller, F., Ponsar, H., and Reif, W. (2019). Ensemble programming for multipotent systems. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*.

Kosak, O., Huhn, L., Bohn, F., Wanninger, C., Hoffmann, A., and Reif, W. (2020b). MapleSwarm: Programming Collective Behavior for Ensembles by Extending HTN-Planning. In *9th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation.*

Kosak, O., Wanninger, C., Angerer, A., Hoffmann, A., Schiendorfer, A., and Seebach, H. (2016a). Towards self-organizing swarms of reconfigurable self-aware robots. In *Found. and Applications of Self* Systems, IEEE Int. Workshops on*, pages 204–209. IEEE.

Kosak, O., Wanninger, C., Angerer, A., Hoffmann, A., Schierl, A., and Seebach, H. (2016b). Decentralized coordination of heterogeneous ensembles using jadex. In *IEEE 1st Int. Workshops on Found. and Appl. of Self* Systems (FAS*W)*, pages 271–272.

Kosak, O., Wanninger, C., Hoffmann, A., Ponsar, H., and Reif, W. (2018). Multipotent systems: Combining planning, self-organization, and reconfiguration in modular robot ensembles. *Sensors*, 19(1).

Schörner, M., Wanninger, C., Hoffmann, A., Kosak, O., Ponsar, H., and Reif, W. (2020). Modeling and execution of coordinated missions in reconfigurable robot ensembles. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 290–293.

Wanninger, C., Alfano, L., Schörner, M., Hoffmann, A., Kosak, O., and Reif, W. (2021). under review - Semantic Plug and Play: An Architecture Combining Linked Data and Reconfigurable Hardware. In *16th IEEE International Conference on Semantic Computing (ICSC)*. IEEE.

Wanninger, C., Eymüller, C., Hoffmann, A., Kosak, O., and Reif, W. (2018). Synthesising Capabilities for Collective Adaptive Systems from Self-Descriptive Hardware Devices - Bridging the Reality Gap. In *8th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation.*

Wolf, B., Chwala, C., Fersch, B., Garvelmann, J., Junkermann, W., Zeeman, M. J., Angerer, A., Adler, B., Beck, C., Brosy, C., Brugger, P., Emeis, S., Dannenmann, M., Roo, F. D., Diaz-Pines, E., Haas, E., Hagen, M., Hajnsek, I., Jacobeit, J., Jagdhuber, T., Kalthoff, N., Kiese, R., Kunstmann, H., Kosak, O., Krieg, R., Malchow, C., Mauder, M., Merz, R., Notarnicola, C., Philipp, A., Reif, W., Reineke, S., Rödiger, T., Ruehr, N., Schäfer, K., Schrön, M., Senatore, A., Shupe, H., Völksch, I., Wanninger, C., Zacharias, S., and Schmid, H. P. (2017). The scalex campaign: Scale-crossing land surface and boundary layer processes in the tereno-prealpine observatory. *Bulletin of the American Meteorological Society*, 98(6):1217–1234.

Bibliography

Alexander Pokahr, L. B. and Jander, K. (2018). Jadex Active Components. *Available at https://www.activecomponents.org/, accessed on 2021-03-17.*

Alting, L. and Zhang, H. (1989). Computer aided process planning: the state-of-the-art survey. *The Int. Journ. of Production Res.*, 27(4):553–585.

Amazon (2020). Amazon Prime Air. *Available at https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011, accessed on 2020-01-13.*

Amigoni, F., Gatti, N., Pinciroli, C., and Roveri, M. (2005). What planner for ambient intelligence applications? *IEEE Trans. on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 35(1):7–21.

Anders, G., Schiendorfer, A., Siefert, F., Steghöfer, J.-P., and Reif, W. (2015). Cooperative Resource Allocation in Open Systems of Systems. *ACM Trans. Auton. Adapt. Syst.*, 10(2):11:1–11:44.

Anders, G., Siefert, F., Schiendorfer, A., Seebach, H., Steghöfer, J.-P., Eberhardinger, B., Kosak, O., and Reif, W. (2016). *Specification and Design of Trust-Based Open Self-Organising Systems*, pages 17–54. Springer Int. Publishing, Cham.

Andersen, T., Scheeren, B., Peters, W., and Chen, H. (2018). A uav-based active aircore system for measurements of greenhouse gases. *Atmospheric Measurement Techniques*, 11(5):2683–2699.

Andreas, D. C., Eckhoff, C. D., and Loveman, R. D. (2005). Serial device daisy chaining method and apparatus. *Available at https://patents.google.com/patent/US20030074505, accessed on 2018-11-15.* US Patent 6,928,501.

Angerer, A., Hoffmann, A., Schierl, A., Vistein, M., and Reif, W. (2013). Robotics api: Object-oriented software development for industrial robots. *Journ. of Software Engineering for Robotics*, 4(1):1–22.

APM (2020). APM Planner 2 official website. *Available at http://ardupilot.org/planner2/, accessed on 2020-01-12.*

Apt, K. (2003). *Principles of constraint programming.* Cambridge University Press.

Ashley-Rollman, M. P., Goldstein, S. C., Lee, P., Mowry, T. C., and Pillai, P. (2007). Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2794–2800.

Ashley-Rollman, M. P., Lee, P., Goldstein, S. C., Pillai, P., and Campbell, J. D. (2009). A language for large ensembles of independently executing nodes. In Hill, P. M. and Warren, D. S., editors, *Logic Programming*, pages 265–280, Berlin, Heidelberg. Springer Berlin Heidelberg.

Autoquad (2018). Autoquad - Autonomous Multi Rotor Vehicle Controller. *Available at `http://autoquad.org/`, accessed on 2018-11-15.*

Awaad, I., Kraetzschmar, G. K., and Hertzberg, J. (2014). Finding ways to get the job done: An affordance-based approach. In *ICAPS*.

Aydin, B., Selvi, E., Tao, J., and Starek, M. J. (2019). Use of fire-extinguishing balls for a conceptual system of drone-assisted wildfire fighting. *Drones*, 3(1).

Babaoglu, O., Shrobe, H., and eds. (2007). First IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007) 9-11 July, Boston, MA, USA. *Available at `http://projects.csail.mit.edu/saso2007/`, accessed on 2018-11-15.*

Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. (1989). Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322.

Banerjee, S. and Hecker, J. P. (2017). A multi-agent system approach to load-balancing and resource allocation for distributed computing. In Bourgine, P., Collet, P., and Parrend, P., editors, *First Complex Systems Digital Campus World E-Conference 2015*, pages 41–54, Cham. Springer International Publishing.

Barca, J. C. and Sekercioglu, Y. A. (2013). Swarm robotics reviewed. *Robotica*, 31(03):345–359.

Barchyn, T. E., Hugenholtz, C. H., Myshak, S., and Bauer, J. (2017). A uav-based system for detecting natural gas leaks. *Journal of Unmanned Vehicle Systems*, 6(1):18–30.

Bartusch, M., Möhring, R. H., and Radermacher, F. J. (1988). Scheduling project networks with resource constraints and time windows. *Annals of Operations Res.*, 16(1):199–240.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., and Turbak, F. (2017). Learnable programming. *Communications of the ACM*, 60(6):72–80.

Beasley, R. A. (2012). Medical robots: current systems and research directions. *Journal of Robotics*, 2012.

Becker, M., Blatt, F., and Szczerbicka, H. (2013). *A Multi-agent Flooding Algorithm for Search and Rescue Operations in Unknown Terrain*, pages 19–28. Springer Berlin Heidelberg, Berlin, Heidelberg.

Behnke, G., Höller, D., and Biundo, S. (2019). Finding optimal solutions in htn planning-a sat-based approach. In *IJCAI*, pages 5500–5508.

Berrahal, S., Kim, J.-H., Rekhis, S., Boudriga, N., Wilkins, D., and Acevedo, J. (2016). Border surveillance monitoring using quadcopter uav-aided wireless sensor networks. *Journal of Communications Software and Systems*.

Bertoli, P. and Cimatti, A. (2002). Improving heuristics for planning as search in belief space. In *AIPS*, pages 143–152.

Bianchi, L., Gambardella, L. M., and Dorigo, M. (2002). An ant colony optimization approach to the probabilistic traveling salesman problem. In *International Conference on Parallel Problem Solving from Nature*, pages 883–892. Springer.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. In Erol et al. [1994], pages 281–300.

Boeing (2019). Watch: Cargo Air Vehicle Completes First Outdoor Flight. *Available at `http://www.boeing.com/features/2019/05/cav-first-flight-05-19.page`, accessed on 2021-02-04*.

Bohn, F. (2018). Implementierung von Selbstorganisationsmechanismen für die Ausführung von ScORe-Missionen in Multi-Robotersystemen. *Bachelors Thesis - not published*.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Art. Intelligence*, 129(1):5 – 33.

Boubaker, O. (2013). The inverted pendulum benchmark in nonlinear control theory: A survey. *International Journal of Advanced Robotic Systems*, 10(5):233.

Bowling, M., Browning, B., and Veloso, M. (2004). Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of the Fourteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'04, page 376–383. AAAI Press.

Brafman, R. I. and Domshlak, C. (2008). From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, pages 28–35.

Brambilla, M., Ferrante, E., Birattari, M., and Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41.

Braubach, L. and Pokahr, A. (2012). Developing distributed systems with active components and jadex. *Scalable Computing: Practice and Experience*, 13(2):100–120.

Breitenmoser, A., Schwager, M., Metzger, J. C., Siegwart, R., and Rus, D. (2010). Voronoi coverage of non-convex environments with a group of networked robots. In *2010 IEEE Intern. Conf. on Robotics and Automation*, pages 4982–4989.

Brenner, C., Zeeman, M., Bernhardt, M., and Schulz, K. (2018). Estimation of evapotranspiration of temperate grassland based on high-resolution thermal and visible range imagery from unmanned aerial systems. *International Journal of Remote Sensing*, 39(15-16):5141–5174. PMID: 30246176.

Brenner, M. and Nebel, B. (2009). Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331.

Brisset, P., Drouin, A., Gorraz, M., Huard, P. S., and Tyler, J. (2006). The Paparazzi Solution. In *MAV 2006, 2nd US-European Competition and Workshop on Micro Air Vehicles*, Sandestin, United States.

Brooks, D. J., Begum, M., and Yanco, H. A. (2016). Analysis of reactions towards failures and recovery strategies for autonomous robots. In *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 487–492.

Brosy, C., Krampf, K., Zeeman, M., Wolf, B., Junkermann, W., Schäfer, K., Emeis, S., and Kunstmann, H. (2017). Simultaneous multicopter-based air sampling and sensing of meteorological variables. *Atmospheric Measurement Techniques*, 10(8):2773–2784.

Brumitt, B. L. and Stentz, A. (1998). Grammps: a generalized mission planner for multiple mobile robots in unstructured environments. In *Robotics and Automation, 1998. Proc. 1998 IEEE Int. Conf. on*, volume 2, pages 1564–1571 vol.2.

Brunner, D. (2018). Maximum climb altitude of a drone. *Available at https://www.technik-consulting.eu/en/analysis/drone.html, accessed on 2021-02-04*.

Brutschy, A., Pini, G., Pinciroli, C., Birattari, M., and Dorigo, M. (2014). Self-organized task allocation to sequentially interdependent tasks in swarm robotics. *Autonomous agents and multi-agent systems*, 28(1):101–125.

Bürkle, A., Segor, F., and Kollmann, M. (2011). Towards autonomous micro uav swarms. *Journal of intelligent & robotic systems*, 61(1-4):339–353.

Cacace, J., Finzi, A., Lippiello, V., Furci, M., Mimmo, N., and Marconi, L. (2016). A control architecture for multiple drones operated via multimodal interaction in search rescue mission. In *2016 IEEE Intern. Symposium on Safety, Security, and Rescue Robotics (SSRR)*, pages 233–239.

Campion, M., Ranganathan, P., and Faruque, S. (2019). Uav swarm communication and control architectures: a review. *Journal of Unmanned Vehicle Systems*, 7(2):93–106.

Caron, D., Stauffer, B., Darjany, L., Oberg, C., Pereira, A., Das, J., Heidarsson, H., Smith, R., Smith, E., Seubert, E., et al. (2009). Networked aquatic microbial observing systems: An overview. *Center for Embedded Network Sensing*.

CCL (2020). Netlogo - northwestern's center for connected learning and computer-based modeling (ccl). *Available at https://ccl.northwestern.edu/netlogo/, accessed on 2020-08-28*.

Çelikkanat, H., Turgut, A. E., and Şahin, E. (2009). *Guiding a Robot Flock via Informed Robots*, pages 215–225. Springer Berlin Heidelberg, Berlin, Heidelberg.

Chevaleyre, Y., Dunne, P. E., Endriss, U., Lang, J., Lemaitre, M., Maudet, N., Padget, J., Phelps, S., Rodriguez-Aguilar, J. A., and Sousa, P. (2006). Issues in multiagent resource allocation. *Informatica*, 30(1).

Choi, H.-J., Kim, Y.-D., and Kim, H.-J. (2011). Genetic algorithm based decentralized task assignment for multiple unmanned aerial vehicles in dynamic environments. *International Journal of Aeronautical and Space Sciences*, 12(2):163–174.

Choi, H.-L., Whitten, A. K., and How, J. P. (2010). Decentralized task allocation for heterogeneous teams with cooperation constraints. In *Proceedings of the 2010 American Control Conference*, pages 3057–3062.

CNBC (2018a). BASF Faces Prolonged Shut-Down After Chemical Site Explosion. *Available at https://www.bloomberg.com/news/articles/2016-10-17/basf-reports-explosion-at-its-biggest-site-in-ludwigshafen, accessed on 2018-11-15.*

CNBC (2018b). Chemical plant explosion thrusts Arkema into spotlight. *Available at https://www.cnbc.com/2017/09/02/chemical-plant-explosion-thrusts-arkema-into-spotlight.html, accessed on 2018-11-15.*

CO., X. S. (2018). Orangepizero website. *Available at http://www.orangepi.org/orangepizero/, accessed on 2018-11-15.*

CockroachLabs (2021). CockroachDB The most highly evolved database on the planet. *Available at https://www.cockroachlabs.com/, accessed on 2021-05-20.*

Colorni, A., Dorigo, M., Maniezzo, V., and Trubian, M. (1994). Ant system for job-shop scheduling. *Belgian Journ. of Operations Res., Statistics and Computer Science*, 34(1):39–53.

Coltin, B. and Veloso, M. (2010). Mobile robot task allocation in hybrid wireless sensor networks. In *Intell. Robots and Systems (IROS), 2010 IEEE/RSJ Int. Conf. on*, pages 2932–2937.

Coltin, B. and Veloso, M. (2013). Online pickup and delivery planning with transfers for mobile robots. In *Workshops at the Twenty-Seventh AAAI Conference on Artificial Intelligence.*

Costelha, H. and Lima, P. (2012). Robot task plan representation by petri nets: modelling, identification, analysis and execution. *Autonomous Robots*, 33(4):337–360.

Cui, Y., Lane, J., Voyles, R., and Krishnamoorthy, A. (2014). A new fault tolerance method for field robotics through a self-adaptation architecture. In *2014 IEEE Int. Symp. on Saf., Sec., and Resc. Robot. (2014)*, pages 1–6. IEEE.

Daniel, K., Dusza, B., Lewandowski, A., and Wietfelds, C. (2009). Airshield: A system-of-systems muav remote sensing architecture for disaster response. In *Proc. 3rd Annual IEEE Systems Conf. (SysCon).*

Daniel, K., Rohde, S., and Wietfeld, C. (2010). Leveraging public wireless communication infrastructures for uav-based sensor networks. In *2010 IEEE Int. Conf. on Technologies for Homeland Security (HST)*, pages 179–184.

Davidson, C. I., Phalen, R. F., and Solomon, P. A. (2005). Airborne particulate matter and human health: a review. *Aerosol Science and Technology*, 39(8):737–749.

Davies, B. and Darbyshire, I. (1984). The use of expert systems in process-planning. *CIRP Annals - Manufacturing Technology*, 33(1):303 – 306.

De Cubber, G., Serrano, D., Berns, K., Chintamani, K., Sabino, R., Ourevitch, S., Doroftei, D., Armbrust, C., Flamma, T., and Baudoin, Y. (2013). Search and rescue robots developed by the european icarus project. In *7th Int. Workshop on Robotics for Risky Environments*. Citeseer.

Dedousis, D. and Kalogeraki, V. (2018). A framework for programming a swarm of uavs. In *Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference*, pages 5–12.

DHL, D. P. G. (2019). DHL Express Launches Its First Regular Fully-Automated And Intelligent Urban Delivery Service. *Available at `https://www.dhl.com/global-en/home/press/press-archive/2019/dhl-express-launches-its-first-regular-fully-automated-and-intelligent-urban-drone-delivery-service.html`, accessed on 2020-01-13*.

Di Marzo Serugendo, G., Foukia, N., Hassas, S., Karageorgos, A., Mostéfaoui, S. K., Rana, O. F., Ulieru, M., Valckenaers, P., and Van Aart, C. (2004). Self-organisation: Paradigms and applications. In Di Marzo Serugendo, G., Karageorgos, A., Rana, O. F., and Zambonelli, F., editors, *Engineering Self-Organising Systems*, pages 1–19, Berlin, Heidelberg. Springer Berlin Heidelberg.

Dias, M. and Stentz, A. (1999). A free market architecture for coordinating multiple robots. Technical report, CMU-RI-TR-99-42, Carnegie Mellon University.

Dias, M. B., Zlot, R., Kalra, N., and Stentz, A. (2006). Market-based multirobot coordination: A survey and analysis. *Proc. of the IEEE*, 94(7):1257–1270.

Do, M. B. and Kambhampati, S. (2001). Planning as constraint satisfaction: Solving the planning graph by compiling it into csp. *Art. Intelligence*, 132(2):151 – 182.

Dominguez, M. H., Nesmachnow, S., and Hernández-Vega, J.-I. (2017). Planning a drone fleet using artificial intelligence for search and rescue missions. In *2017 IEEE XXIV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, pages 1–4. IEEE.

Dorigo, M., Birattari, M., and Stutzle, T. (2006). Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39.

Dorigo, M., Floreano, D., Gambardella, L. M., Mondada, F., Nolfi, S., Baaboura, T., Birattari, M., Bonani, M., Brambilla, M., Brutschy, A., Burnier, D., Campo, A., Christensen, A. L., Decugniere, A., Caro, G. D., Ducatelle, F., Ferrante, E., Forster, A., Gonzales, J. M., Guzzi, J., Longchamp, V., Magnenat, S., Mathews, N., de Oca, M. M., O'Grady, R., Pinciroli, C., Pini, G., Retornaz, P., Roberts, J., Sperati, V., Stirling, T., Stranieri, A., Stutzle, T., Trianni, V., Tuci, E., Turgut, A. E., and Vaussard, F. (2013). Swarmanoid: A novel concept for the study of heterogeneous robotic swarms. *IEEE RAM*, 20(4):60–71.

Dorigo, M., Tuci, E., Groß, R., Trianni, V., Labella, T. H., Nouyan, S., Ampatzis, C., Deneubourg, J.-L., Baldassarre, G., Nolfi, S., et al. (2004). The swarm-bots project. In *Int. Workshop on Swarm Robotics*, pages 31–44. Springer.

Dousse, N., Heitz, G., and Floreano, D. (2016). Extension of a ground control interface for swarms of small drones. *Artificial Life and Robotics*, 21(3):308–316.

Dronecode (2019). QGroundControl - Intuitive and Powerful Ground Control Station for the MAVLINK protocol. *Available at `http://qgroundcontrol.com/`, accessed on 2021-03-17.*

Duarte, M., Costa, V., Gomes, J., Rodrigues, T., Silva, F., Oliveira, S. M., and Christensen, A. L. (2016). Evolution of collective behaviors for a real swarm of aquatic surface robots. *PLoS ONE*, 11(3):1–25.

Dunbabin, M. and Marques, L. (2012). Robots for environmental monitoring: Significant advancements and applications. *IEEE Robotics Automation Magazine*, 19(1):24–39.

Dyck, S. and Peschke, G. (1983). *Grundlagen der Hydrologie*. Ernst Berlin.

Ed Durfree, S. Z. (2013). *Multiagent Planning, Control, and Execution*, chapter 11, pages 485–545. MIT press.

Eing, L. (2020). Integration domänenspezifischer Sprachen in multipotente Systeme am Beispiel von Protelis. *Bachelors Thesis - not published.*

Elkawkagy, M. and Biundo, S. (2011). *Hybrid Multi-agent Planning*, pages 16–28. Springer Berlin Heidelberg, Berlin, Heidelberg.

Entrop, A. and Vasenev, A. (2017). Infrared drones in the construction industry: designing a protocol for building thermography procedures. *Energy Procedia*, 132:63 – 68. 11th Nordic Symposium on Building Physics, NSB2017, 11-14 June 2017, Trondheim, Norway.

Erol, K., Hendler, J., and Nau, D. S. (1994). Htn planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128.

Escobedo, F. J. and Nowak, D. J. (2009). Spatial heterogeneity and air pollution removal by an urban forest. *Landscape and Urban Planning*, 90(3):102 – 110.

Eskicioglu, H. and Davies, B. (1983). An interactive process planning system for prismatic parts (icapp). *CIRP Annals - Manufacturing Technology*, 32(1):365 – 370.

Euronews (2018). Eight injured in German oil refinery explosion. *Available at `https://www.euronews.com/2018/09/01/eight-injured-in-german-oil-refinery-explosion`, accessed on 2018-11-15.*

European Commission (2017). Chemical Accident Prevention and Preparedness. *Available at `https://www.oecd.org/chemicalsafety/chemical-accidents/guiding-principles-chemical-accident-prevention-preparedness-and-response.htm`, accessed on 2021-02-04.*

Eymüller, C., Wanninger, C., Hoffmann, A., and Reif, W. (2018). Semantic Plug and Play – Self-Descriptive Modular Hardware for Robotic Applications. In *International Journal of Semantic Computing (IJSC)*.

Faci, N., Guessoum, Z., and Marin, O. (2006). Dimax: a fault-tolerant multi-agent platform. In *Proc. of the 2006 Intern. Work. on Software engineering for large-scale multi-agent Sys.*, pages 13–20. ACM.

Fersch, B., Francke, T., Heistermann, M., Schrön, M., Döpper, V., Jakobi, J., Baroni, G., Blume, T., Bogena, H., Budach, C., et al. (2020). A dense network of cosmic-ray neutron sensors for soil moisture observation in a highly instrumented pre-alpine headwater catchment in germany. *Earth System Science Data*, 12(3):2289–2309.

Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Art. intelligence*, 2(3-4):189–208.

Fink*, E. and Blythe, J. (2005). Prodigy bidirectional planning. *Journ. of Experimental & Theoretical Art. Intelligence*, 17(3):161–200.

Fioretto, F., Pontelli, E., and Yeoh, W. (2016). Distributed constraint optimization problems and applications: A survey. *CoRR*.

Fioretto, F., Pontelli, E., and Yeoh, W. (2018). Distributed constraint optimization problems and applications: A survey. *Journal of Artificial Intelligence Research*, 61:623–698.

Flushing, E. F., Gambardella, L. M., and Caro, G. A. D. (2014). A mathematical programming approach to collaborative missions with heterogeneous teams. In *2014 IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, pages 396–403.

Furley, D. and Cole, T. (1970). Democritus and the sources of greek anthropology. *Journal of Hellenic Studies*, 90:239.

Gade, S. and Joshi, A. (2013). Heterogeneous uav swarm system for target search in adversarial environment. In *2013 International Conference on Control Communication and Computing (ICCC)*, pages 358–363. IEEE.

Gancet, J., Hattenberger, G., Alami, R., and Lacroix, S. (2005). Task planning and control for a multi-uav system: architecture and algorithms. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1017–1022. IEEE.

García, P., Caamaño, P., Duro, R. J., and Bellas, F. (2013). Scalable task assignment for heterogeneous multi-robot teams. *International Journal of Advanced Robotic Systems*, 10(2):105.

Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., and Steenkiste, P. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.

Georgeff, M. P. (1984). A theory of action for multiagent planning. In *AAAI*, volume 84, pages 121–125.

Georgievski, I. and Aiello, M. (2014). An overview of hierarchical task network planning. *CoRR*, abs/1403.7426.

Georgievski, I. and Aiello, M. (2015). Htn planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124 – 156.

Gerevini, A. and Long, D. (2006). Preferences and soft constraints in pddl3. In Gerevini, A. and Long, D., editors, *ICAPS Workshop on Planning with Preferences and Soft Constraints*, pages 46–53.

Gerkey, B. P. and Matarić, M. J. (2004). A formal analysis and taxonomy of task allocation in multi-robot systems. *The Int. Journ. of Robotics Res.*, 23(9):939–954.

Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated planning: theory & practice*. Elsevier.

Ghamry, K. A., Kamel, M. A., and Zhang, Y. (2017). Multiple uavs in forest fire fighting mission using particle swarm optimization. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1404–1409. IEEE.

Gharehchopogh, F. S. and Arjang, H. (2014). A survey and taxonomy of leader election algorithms in distributed systems. *Indian Journ. of Science and Technology*, 7(6):815.

Ghosh, S. (2014). *Distributed systems: an algorithmic approach*. Chapman and Hall/CRC.

Giorgetti, A., Marchetto, M. C., Li, M., Yu, D., Fazzina, R., Mu, Y., Adamo, A., Paramonov, I., Cardoso, J. C., Monasterio, M. B., et al. (2012). Cord blood-derived neuronal cells by ectopic expression of sox2 and c-myc. *Proceedings of the National Academy of Sciences*, 109(31):12556–12561.

Goldstein, J. (1999). Emergence as a construct: History and issues. *Emergence*, 1(1):49–72.

Gorniak, P. and Davis, I. (2007). Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters. In *AIIDE*, pages 14–19.

Greenpeace (2006). Die Feinstaub-Vorhersagekarte. *Available at `https://www.greenpeace.de/themen/endlager-umwelt/die-feinstaub-vorhersagekarte`, accessed on 2021-02-04.*

GROSENS INSTRUMENTS GmbH (2018). Temod. *Available at `https://www.mikrocontroller.net/attachment/46744/I2C_Temperaturmodul_DBD.pdf`, accessed on 2018-11-15.*

Gu, J., Su, T., Wang, Q., Du, X., and Guizani, M. (2018). Multiple moving targets surveillance based on a cooperative network for multi-uav. *IEEE Communications Magazine*, 56(4):82–89.

Gutmann, M. and Rinner, B. (2021). Mission specification and execution of multidrone systems. *Not Published. Available at `https://bernhardrinner.com/pubs/2021/Gutmann_DATE2021.pdf`, accessed on 2020-01-13.*

Hamann, H., Khaluf, Y., Botev, J., Divband Soorati, M., Ferrante, E., Kosak, O., Montanier, J.-M., Mostaghim, S., Redpath, R., Timmis, J., Veenstra, F., Wahby, M., and Zamuda, A. (2016). Hybrid societies: Challenges and perspectives in the design of collective behavior in self-organizing systems. *Frontiers in Robotics and AI*, 3:14.

Hanke, J., Kosak, O., Schiendorfer, A., and Reif, W. (2018). Self-organized resource allocation for reconfigurable robot ensembles. In *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 110–119.

Hardkernel (2018). Odroid xu 4. *Available at* `https://www.hardkernel.com/main/products/prdt_info.php`, *accessed on 2018-11-15*.

Hartmann, A. and Heinze, J. (2003). Lay eggs, live longer: division of labor and life span in a clonal ant species. *Evolution*, 57(10):2424–2429.

Haslum, P. (2006). Improving heuristics through relaxed search. *Journ. of Art. Res.*, 25:233–267.

Heiligenberg, W. (2012). *Principles of electrolocation and jamming avoidance in electric fish: a neuroethological approach*, volume 1. Springer Science & Business Media.

Hernandez, G., Berry, T.-A., Wallis, S., and Poyner, D. (2017). Temperature and humidity effects on particulate matter concentrations in a sub-tropical climate during winter. In *Proceedings of International Conference of the Environment, Chemistry and Biology*. International Association of Computer Science and Information Technology.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Higgins, C., Wing, M., Kelley, J., Sayde, C., Burnett, J., and Holmes, H. (2018). A high resolution measurement of the morning abl transition using distributed temperature sensing and an unmanned aircraft system. *Environmental Fluid Mechanics*, 18(3):683–693.

Higuera, J. C. G. and Dudek, G. (2013). Fair subdivision of multi-robot tasks. In *Robotics and Automation (ICRA), 2013 IEEE Int. Conf. on*, pages 3014–3019.

Hoffmann, A., Angerer, A., Ortmeier, F., Vistein, M., and Reif, W. (2009). Hiding real-time: A new approach for the software development of industrial robots. In *IROS*, pages 2108–2113.

Hoffmann, J. and Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journ. of Art. Intelligence Res.*, 14:253–302.

Holland, R. A., Waters, D. A., and Rayner, J. M. (2004). Echolocation signal structure in the megachiropteran bat rousettus aegyptiacus geoffroy 1810. *Journal of experimental biology*, 207(25):4361–4369.

Hood, S., Benson, K., Hamod, P., Madison, D., O'Kane, J. M., and Rekleitis, I. (2017). Bird's eye view: Cooperative exploration by ugv and uav. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 247–255.

Huang, D., Ma, X., and Zhang, S. (2020). Performance analysis of the raft consensus algorithm for private blockchains. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(1):172–181.

Hudziak, M., Pozniak-Koszalka, I., Koszalka, L., and Kasprzak, A. (2015). Comparison of algorithms for multi-agent pathfinding in crowded environment. In Nguyen, N. T., Trawiński, B., and Kosala, R., editors, *Intel. Inform. and Database Sys.*, pages 229–238, Cham. Springer International Publishing.

Humphreys, T. (2013). Exploring htn planners through examples. *Game AI pro: Collected wisdom of game AI professionals*, 149.

Hussain, H., Malik, S. U. R., Hameed, A., Khan, S. U., Bickler, G., Min-Allah, N., Qureshi, M. B., Zhang, L., Yongji, W., Ghani, N., Kolodziej, J., Zomaya, A. Y., Xu, C.-Z., Balaji, P., Vishnu, A., Pinel, F., Pecero, J. E., Kliazovich, D., Bouvry, P., Li, H., Wang, L., Chen, D., and Rayes, A. (2013). A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*, 39(11):709–736.

Hussein, A., Adel, M., Bakr, M., Shehata, O. M., and Khamis, A. (2014). Multi-robot task allocation for search and rescue missions. *Journ. of Physics: Conf. Series*, 570(5):052006.

Intel (2020). Intel Shooting Star drone project page. *Available at `https://www.intel.co.uk/content/www/uk/en/technology-innovation/aerial-technology-light-show.html`, accessed on 2020-01-12.*

Intel (2021). Aerial Technology Light Show. *Available at `https://www.intel.de/content/www/de/de/technology-innovation/aerial-technology-light-show.html`, accessed on 2021-02-01.*

Iordache, G. V., Boboila, M. S., Pop, F., Stratan, C., and Cristea, V. (2007). A decentralized strategy for genetic scheduling in heterogeneous environments. *Multiagent and Grid Systems*, 3(4):355–367.

ISSELabs (2018). Flying robot ensemble in action at the ScaleX 2016 geographic measurement campaign. *Available at `https://youtu.be/MWNyUymtNSs`, accessed on 2018-11-15.*

Jakobs, H. J., Friese, E., Memmesheimer, M., and Ebel, A. (2002). A real-time forecast system for air pollution concentrations. In *Proceedings of EUROTRAC Symposium 2002.*

Jayaratne, R., Liu, X., Thai, P., Dunbabin, M., and Morawska, L. (2018). The influence of humidity on the performance of a low-cost air particle mass sensor and the effect of atmospheric fog. *Atmospheric Measurement Techniques*, 11(8):4883–4890.

Jevtic, A., Gutierrez, l., Andina, D., and Jamshidi, M. (2012). Distributed bees algorithm for task allocation in swarm of robots. *IEEE Systems Journal*, 6(2):296–304.

Kaimal, J. C. and Finnigan, J. J. (1994). *Atmospheric boundary layer flows: their structure and measurement.* Oxford university press.

Kamei, K., Nishio, S., Hagita, N., and Sato, M. (2012). Cloud networked robotics. *IEEE Network*, 26(3):28–34.

Kautz, H. and Selman, B. (2006). Satplan04: Planning as satisfiability. *Working Notes on the Fifth Int. Planning Competition*, pages 45–46.

Kelly, J.-P., Botea, A., Koenig, S., et al. (2007). Planning with hierarchical task networks in video games. In *Proceedings of the ICAPS-07 Workshop on Planning in Games*.

Kelly, J. P., Botea, A., Koenig, S., et al. (2008). Offline planning with hierarchical task networks in video games. In *AIIDE*, pages 60–65.

Khaluf, Y. (2016). Adaptive construction behavior in robot swarm. *Available at https://www.semanticscholar.org/paper/Adaptive-Construction-Behavior-in-Robot-Swarms-Khaluf/29ee2ffb9e2e318895e88ba625fc970bf8cd8284?p2df, accessed on 2018-11-15*.

Khamis, A., Hussein, A., and Elmogy, A. (2015). Multi-robot task allocation: A review of the state-of-the-art. In *Cooperative Robots and Sensor Networks 2015*, pages 31–51. Springer.

Khamis, A. M., Elmogy, A. M., and Karray, F. O. (2011). Complex task allocation in mobile surveillance systems. *Journ. of Intel. & Robotic Systems*, 64(1):33–55.

KIT IMK/IFU, G.-P. (2018). Scalex. *Available at https://scalex.imk-ifu.kit.edu/, accessed on 2018-11-15*.

Koenig, S. (2001). Agent-centered search. *AI Magazine*, 22(4):109.

Kolarić, D., Skala, K., and Dubravić, A. (2008). Integrated system for forest fire early detection and management. *Periodicum biologorum*, 110(2):205–211.

Kolisch, R. and Hartmann, S. (1999). *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pages 147–178. Springer US, Boston, MA.

Koppensteiner, G., Merdan, M., Lepuschitz, W., and Hegny, I. (2009). Hybrid based approach for fault tolrance in a multi-agent system. In *2009 IEEE/ASME Intern. Conf. on Adv. Intel. Mechatronics*, pages 679–684. IEEE.

Koren, Y. and Borenstein, J. (1991). Potential field methods and their inherent limitations for mobile robot navigation. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1398–1404. IEEE.

Korsah, G. A., Stentz, A., and Dias, M. B. (2013). A comprehensive taxonomy for multi-robot task allocation. *The Int. Journ. of Robotics Res.*, 32(12):1495–1512.

Kosak, O. (2015). A decentralised swarm approach for mobile robot-systems. In *Organic Computing: Doctoral Dissertation Colloquium 2015*, volume 7, page 53. kassel university press GmbH.

Kosak, O. (2017). Facilitating planning by using self-organization. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 371–374.

Kosak, O. (2018). Multipotent systems: A new paradigm for multi-robot applications. In *Organic Computing: Doctoral Dissertation Colloquium 2018*, volume 10, page 53. kassel university press GmbH.

Kosak, O., Anders, G., Siefert, F., and Reif, W. (2015). An Approach to Robust Resource Allocation in Large-Scale Systems of Systems. In *Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th Int. Conf. on*, pages 1–10.

Kosak, O., Bohn, F., Eing, L., Rall, D., Wanninger, C., Hoffmann, A., and Reif, W. (2020a). Swarm and Collective Capabilities for Multipotent Robot Ensembles, currently under review. In *9th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation.*

Kosak, O., Bohn, F., Keller, F., Ponsar, H., and Reif, W. (2019). Ensemble programming for multipotent systems. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W).*

Kosak, O., Huhn, L., Bohn, F., Wanninger, C., Hoffmann, A., and Reif, W. (2020b). MapleSwarm: Programming Collective Behavior for Ensembles by Extending HTN-Planning. In *9th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation.*

Kosak, O., Wanninger, C., Angerer, A., Hoffmann, A., Schiendorfer, A., and Seebach, H. (2016a). Towards self-organizing swarms of reconfigurable self-aware robots. In *Found. and Applications of Self* Systems, IEEE Int. Workshops on*, pages 204–209. IEEE.

Kosak, O., Wanninger, C., Angerer, A., Hoffmann, A., Schierl, A., and Seebach, H. (2016b). Decentralized coordination of heterogeneous ensembles using jadex. In *IEEE 1st Int. Workshops on Found. and Appl. of Self* Systems (FAS*W)*, pages 271–272.

Kosak, O., Wanninger, C., Hoffmann, A., Ponsar, H., and Reif, W. (2018). Multipotent systems: Combining planning, self-organization, and reconfiguration in modular robot ensembles. *Sensors*, 19(1).

Koutsoubelias, M. and Lalis, S. (2016). Tecola: A programming framework for dynamic and heterogeneous robotic teams. In *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, pages 115–124.

Krauter, K., Buyya, R., and Maheswaran, M. (2002). A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164.

KUKA (2018). KUKA Youbot. *Available at* `https://spectrum.ieee.org/automaton/robotics/industrial-robots/scoop-kukas-youbot`, *accessed on 2018-11-15.*

Laborie, P. (2003). Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Art. Intelligence*, 143(2):151 – 188.

Lacroix, S., Alami, R., Lemaire, T., Hattenberger, G., and Gancet, J. (2007). *Decision Making in Multi-UAVs Systems: Architecture and Algorithms*, pages 15–48. Springer Berlin Heidelberg, Berlin, Heidelberg.

Lai, K., Huberman, B. A., and Fine, L. R. (2004). Tycoon: A distributed market-based resource allocation system. *CoRR*, cs.DC/0404013.

Landolsi, T., Sagahyroon, A., Mirza, M., Aref, O., Maki, F., and Maki, S. (2018). Pollution monitoring system using position-aware drones with 802.11 ad-hoc networks. In *2018 IEEE Conference on Wireless Sensors (ICWiSe)*, pages 40–43. IEEE.

LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

Lawler, E. L., Lenstra, J. K., Kan, A. H. R., and Shmoys, D. B. (1993). Chapter 9 sequencing and scheduling: Algorithms and complexity. In *Logistics of Production and Inventory*, volume 4 of *Handbooks in Operations Res. and Management Science*, pages 445 – 522. Elsevier.

Lee, K.-B., Kim, Y.-J., and Hong, Y.-D. (2018). Real-time swarm search method for real-world quadcopter drones. *Applied Sciences*, 8(7):1169.

Lewyckyj, N., Biesemans, J., and Everaerts, J. (2007). OSIRIS: A european project using a high altitude platform for forest fire monitoring. In *Safety and Security Engineering II*, volume 94 of *WIT Trans. on The Built Environment*, pages 205–213. WIT Press.

Li, C. J. and Ling, H. (2015). Synthetic aperture radar imaging using a small consumer drone. In *2015 IEEE International Symposium on Antennas and Propagation USNC/URSI National Radio Science Meeting*, pages 685–686.

Li, X., Ercan, M. F., and Fung, Y. F. (2009). A triangular formation strategy for collective behaviors of robot swarm. In Gervasi, O., Taniar, D., Murgante, B., Laganà, A., Mun, Y., and Gavrilova, M. L., editors, *Computational Science and Its Applications – ICCSA 2009*, pages 897–911, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lima, K., Marques, E. R., Pinto, J., and Sousa, J. B. (2018). Dolphin: a task orchestration language for autonomous vehicle networks. In *IEEE/RSJ Int. Conf. on Intel. Robots and Systems (IROS)*, pages 603–610. IEEE.

Lin, M., Chen, Y., Burnett, R. T., Villeneuve, P. J., and Krewski, D. (2002). The influence of ambient coarse particulate matter on asthma hospitalization in children: case-crossover and time-series analyses. *Environmental health perspectives*, 110(6):575–581.

Liu, C. and Kroll, A. (2012). A centralized multi-robot task allocation for industrial plant inspection by using a* and genetic algorithms. In Rutkowski, L., Korytkowski, M., Scherer, R., Tadeusiewicz, R., Zadeh, L. A., and Zurada, J. M., editors, *Artificial Intelligence and Soft Computing*, pages 466–474, Berlin, Heidelberg. Springer Berlin Heidelberg.

Liu, Y., Liu, H., Tian, Y., and Sun, C. (2020). Reinforcement learning based two-level control framework of uav swarm for cooperative persistent surveillance in an unknown urban area. *Aerospace Science and Technology*, 98:105671.

Loh, E. (2018). Medicine and the rise of the robots: a qualitative review of recent advances of artificial intelligence in health. *BMJ Leader*.

Ma, M. and Yang, Y. (2007). Adaptive triangular deployment algorithm for unattended mobile sensor networks. *IEEE Transactions on Computers*, 56(7):946–847.

Magnenat, S., Chappelier, J.-C., and Mondada, F. (2012). Integration of online learning into htn planning for robotic tasks. In *AAAI Spring Symposium: Designing Intell. Robots*.

Magnenat, S., Voelkle, M., and Mondada, F. (2009). Planner9, a HTN planner distributed on groups of miniature mobile robots. In *Intell. Robotics and Applications, Proceedings of the Second International Conference on Intelligent Robotics and Application*, volume 5928 of *Lecture Notes in Computer Science*, pages 1013–1022. Springer.

Maher, M. and Puget, J.-F. (2003). *Principles and Practice of Constraint Programming-CP98: 4th International Conference, CP98, Pisa, Italy, October 26-30, 1998, Proceedings*. Springer.

Malaschuk, O. and Dyumin, A. (2020). Intelligent multi-agent system for rescue missions. In Misyurin, S. Y., Arakelian, V., and Avetisyan, A. I., editors, *Advanced Technologies in Robotics and Intelligent Systems*, pages 89–97, Cham. Springer International Publishing.

Manyam, S. G., Rasmussen, S., Casbeer, D. W., Kalyanam, K., and Manickam, S. (2017). Multi-uav routing for persistent intelligence surveillance reconnaissance missions. In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 573–580.

Marconi, L., Leutenegger, S., Lynen, S., Burri, M., Naldi, R., and Melchiorri, C. (2013). Ground and aerial robots as an aid to alpine search and rescue: Initial sherpa outcomes. In *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE Int. symposium on*, pages 1–2. IEEE.

Marconi, L., Melchiorri, C., Beetz, M., Pangercic, D., Siegwart, R., Leutenegger, S., Carloni, R., Stramigioli, S., Bruyninckx, H., Doherty, P., Kleiner, A., Lippiello, V., Finzi, A., Siciliano, B., Sala, A., and Tomatis, N. (2012). 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)*, pages 1–4.

Martinez, C., Sampedro, C., Chauhan, A., and Campoy, P. (2014). Towards autonomous detection and tracking of electric towers for aerial power line inspection. In *2014 Intern. Conf. on Unmanned Aircraft Systems (ICUAS)*, pages 284–295.

Martinez-de Dios, J. R., Arrue, B. C., Ollero, A., Merino, L., and Gómez-Rodríguez, F. (2008). Computer vision techniques for forest fire perception. *Image and vision computing*, 26(4):550–562.

Matikainen, L., Lehtomäki, M., Ahokas, E., Hyyppä, J., Karjalainen, M., Jaakkola, A., Kukko, A., and Heinonen, T. (2016). Remote sensing methods for power line corridor surveys. *ISPRS Journ. of Photogrammetry and Remote Sensing*, 119(Supplement C):10 – 31.

McDermott, D. V. (1996). A heuristic estimator for means-ends analysis in planning. In *AIPS*, volume 96, pages 142–149.

Meng, X., Wang, W., and Leong, B. (2015). Skystitch: A cooperative multi-uav-based real-time video surveillance system with stitching. In *Proceedings of the 23rd ACM International Conference on Multimedia*, MM '15, page 261–270, New York, NY, USA. Association for Computing Machinery.

Menif, A., Jacopin, É., and Cazenave, T. (2014). Shpe: Htn planning for video games. In Cazenave, T., Winands, M. H. M., and Björnsson, Y., editors, *Computer Games*, pages 119–132, Cham. Springer International Publishing.

Menssen, S. (2019). A Self-Organization Approach for Robust Task Execution in Multipotent Systems. *Masters Thesis - not published.*

Meseguer, P., Rossi, F., and Schiex, T. (2006). Soft Constraints. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, chapter 9. Elsevier.

Müller, A. (2018). Neue Zürcher Zeitung - Die Blutproben aus dem Zürcher Universitätsspital gelangen jetzt per Drohne ins Labor. *Available at https://www.nzz.ch/zuerich/die-blutproben-aus-dem-zuercher-universitaetsspital-gelangen-jetzt-per-drohne-ins-labor-ld.1441774, accessed on 2020-01-13.*

Momont, A. (2020). Ambulance Drone. *Available at https://www.tudelft.nl/en/ide/research/research-labs/applied-labs/ambulance-drone/, accessed on 2020-01-13.*

Mondada, F., Gambardella, L. M., Floreano, D., Nolfi, S., Deneuborg, J. L., and Dorigo, M. (2005). The cooperation of swarm-bots: physical interactions in collective robotics. *IEEE Robotics Automation Magazine*, 12(2):21–28.

Morgan, D., Subramanian, G. P., Chung, S.-J., and Hadaegh, F. Y. (2016). Swarm assignment and trajectory optimization using variable-swarm, distributed auction assignment and sequential convex programming. *The Int. Journ. of Robotics Res.*, 35(10):1261–1285.

Mosteo, A. R. and Montano, L. (2010). A survey of multi-robot task allocation. *Instituto de Investigacin en Ingenierła de Aragn (I3A), Tech. Rep.*

Mottola, L., Moretta, M., Whitehouse, K., and Ghezzi, C. (2014). Team-level programming of drone sensor networks. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pages 177–190.

Müller-Schloer, C., Schmeck, H., and Ungerer, T. (2011). *Organic computing—a paradigm shift for complex systems.* Springer Science & Business Media.

Murphy, R. R., Tadokoro, S., Nardi, D., Jacoff, A., Fiorini, P., Choset, H., and Erkmen, A. M. (2008). *Search and Rescue Robotics*, pages 1151–1173. Springer Berlin Heidelberg, Berlin, Heidelberg.

Muscettola, N. (2002). *Computing the Envelope for Stepwise-Constant Resource Allocations*, pages 139–154. Springer Berlin Heidelberg, Berlin, Heidelberg.

Myers, K. L. (1999). Cpef: A continuous planning and execution framework. *AI Magazine*, 20(4):63.

Na, H. J. and Yoo, S. (2019). Pso-based dynamic uav positioning algorithm for sensing information acquisition in wireless sensor networks. *IEEE Access*, 7:77499–77513.

Nakamura, K. (2018). Thermoregulatory behavior and its central circuit mechanism-what thermosensory pathway drives it? *Clinical calcium*, 28(1):65–72.

Nau, D. (2013). Game applications of htn planning with state variables. In *Planning in Games: Papers from the ICAPS Workshop.*

Nau, D. S., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., and Yaman, F. (2003). Shop2: An htn planning system. *J. Artif. Intell. Res.(JAIR)*, 20:379–404.

Nedjah, N. and Junior, L. S. (2019). Review of methodologies and tasks in swarm robotics towards standardization. *Swarm and Evolutionary Computation*, 50:100565.

Nedjati, A., Vizvari, B., and Izbirak, G. (2016). Post-earthquake response by small uav helicopters. *Natural Hazards*, 80(3):1669–1688.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). Minizinc: Towards a standard cp modelling language. In Bessière, C., editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg. Springer Berlin Heidelberg.

Neufeld, X., Mostaghim, S., and Perez-Liebana, D. (2017). Htn fighter: Planning in a highly-dynamic game. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, pages 189–194. IEEE.

Newell, A., Simon, H. A., et al. (1959). *Human problem solving*. Prentice-Hall Englewood Cliffs, NJ.

Nguyen, X. and Kambhampati, S. (2001). Reviving partial order planning. In *IJCAI*, volume 1, pages 459–464.

Nissim, R., Brafman, R. I., and Domshlak, C. (2010). A general, fully distributed multi-agent planning algorithm. In *Proc. of the 9th Intern. Conf. on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, pages 1323–1330. Intern. Foundation for Autonomous Agents and Multiagent Systems.

NORDIC SEMICONDUCTORS (2018). Ultra Low Power Wireless Solutions from NORDIC SEMICONDUCTOR. *Available at `https://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01`, accessed on 2018-11-15*.

Obst, O. and Boedecker, J. (2006). *Flexible Coordination of Multiagent Team Behavior Using HTN Planning*, pages 521–528. Springer Berlin Heidelberg, Berlin, Heidelberg.

Østergaard, E. H., Kassow, K., Beck, R., and Lund, H. H. (2006). Design of the atron lattice-based self-reconfigurable robot. *Autonomous Robots*, 21(2):165–183.

Palomaki, R. T., Rose, N. T., van den Bossche, M., Sherman, T. J., and De Wekker, S. F. (2017). Wind estimation in the lower atmosphere using multirotor aircraft. *Journal of Atmospheric and Oceanic Technology*, 34(5):1183–1191.

Patel, R., Rudnick-Cohen, E., Azarm, S., Otte, M., Xu, H., and Herrmann, J. W. (2020). Decentralized task allocation in multi-agent systems using a decentralized genetic algorithm. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3770–3776.

Perez, D., Maza, I., Caballero, F., Scarlatti, D., Casado, E., and Ollero, A. (2013). A ground control station for a multi-uav surveillance system. *Journal of Intelligent & Robotic Systems*, 69(1-4):119–130.

Pérez, I. F., Boumaza, A., and Charpillet, F. (2017). Learning collaborative foraging in a swarm of robots using embodied evolution. In *Artificial Life Conference Proceedings 14*, pages 162–161. MIT Press.

Pianini, D., Viroli, M., and Beal, J. (2015). Protelis: Practical aggregate programming. In *Proc. of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1846–1853. ACM.

Pietzsch, T. (2018). rosewhite. *Available at https://www.rosewhite.de/, accessed on 2018-11-15*.

Pinciroli, C. and Beltrame, G. (2016). Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ Int. Conf. on Intel. Robots and Systems (IROS)*, pages 3794–3800.

Pinedo, M. L. (2016). *Scheduling: Job Shops (Deterministic)*, chapter 6, pages 183–220. Springer Int. Publishing.

Platform, H. O. S. E. (2021). Hc-sr04 ultrasonic sensor module user guide. *Available at https://www.handsontec.com/dataspecs/HC-SR04-Ultrasonic.pdf, accessed on 2021-04-01*.

Pojda, J., Wolff, A., Sbeiti, M., and Wietfeld, C. (2011). Performance analysis of mesh routing protocols for uav swarming applications. In *2011 8th International Symposium on Wireless Communication Systems*, pages 317–321.

Pöschl, U. (2005). Atmospheric aerosols: composition, transformation, climate and health effects. *Angewandte Chemie International Edition*, 44(46):7520–7540.

Preece, A., Gomez, M., de Mel, G., Vasconcelos, W., Sleeman, D., Colley, S., Pearson, G., Pham, T., and La Porta, T. (2008). Matching sensors to missions using a knowledge-based approach. *Proc. of SPIE: Defense Transformation and Net-Centric Systems*, 6981:698109–1.

Pynadath, D. V. and Tambe, M. (2003). An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems*, 7(1):71–100.

QGroundControl (2020). QGroundControl documentation: PlanView. *Available at https://docs.qgroundcontrol.com/en/PlanView/PlanView.html, accessed on 2020-01-12*.

Qin, H., Cui, J. Q., Li, J., Bi, Y., Lan, M., Shan, M., Liu, W., Wang, K., Lin, F., Zhang, Y., et al. (2016). Design and implementation of an unmanned aerial vehicle for autonomous firefighting missions. In *2016 12th IEEE International Conference on Control and Automation (ICCA)*, pages 62–67. IEEE.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe.

Rall, D. (2019). Adaption von Schwarmalgorithmen zum Einsatz in Multi-Roboter Systemen. *Bachelors Thesis - not published*.

Rao, A. S., Georgeff, M. P., et al. (1995). Bdi agents: From theory to practice. In *ICMAS*, volume 95, pages 312–319.

Raspberry (2021). Raspberry Pi 4 Model B Specifications. *Available at https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/, accessed on 2021-02-01.*

Rathinam, S., Sengupta, R., and Darbha, S. (2007). A resource allocation algorithm for multivehicle systems with nonholonomic constraints. *IEEE Transactions on Automation Science and Engineering*, 4(1):98–104.

Read, D. (2006). Auto motion: Robot guidance for manufacturing. US Patent App. 10/502,003.

Restás, Á. (2014). Thematic division and tactical analysis of the uas application supporting forest fire management.

Restas, A. et al. (2015). Drone applications for supporting disaster management. *World Journal of Engineering and Technology*, 3(03):316.

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH computer graphics*, 21(4):25–34.

Richter, S. and Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *Journ. of Art. Intelligence Research*, 39(1):127–177.

RIU, R. (2018). Air Quality Forecast. *Available at http://www.uni-koeln.de/math-nat-fak/geomet/eurad/index_e.html, accessed on 2021-02-04.*

Rizk, Y., Awad, M., and Tunstel, E. W. (2019). Cooperative heterogeneous multi-robot systems: A survey. *ACM Computing Surveys (CSUR)*, 52(2):1–31.

Roberts, R. J. (2020). Nucleic acid. Encyclopedia Britannica. *Available at https://www.britannica.com/science/nucleic-acid, accessed on 2020-06-14.*

Robotics, I. (2018). Innok Heros Modular Versatile Robot System. *Available at https://www.innok-robotics.de/en/products/heros.*

Roldán, J. J., Joossen, G., Sanz, D., Del Cerro, J., and Barrientos, A. (2015). Mini-uav based sensory system for measuring environmental variables in greenhouses. *Sensors*, 15(2):3334–3350.

Roldán, J. J., Lansac, B., del Cerro, J., and Barrientos, A. (2016). A proposal of multi-uav mission coordination and control architecture. In Reis, L. P., Moreira, A. P., Lima, P. U., Montano, L., and Muñoz-Martinez, V., editors, *Robot 2015: Second Iberian Robotics Conference*, pages 597–608, Cham. Springer International Publishing.

Roldán, J. J., del Cerro, J., and Barrientos, A. (2015). A proposal of methodology for multi-uav mission modeling. In *2015 23rd Mediterranean Conference on Control and Automation (MED)*, pages 1–7.

Roldán-Gómez, J. J., González-Gironda, E., and Barrientos, A. (2021). A survey on robotic technologies for forest firefighting: Applying drone swarms to improve firefighters' efficiency and safety. *Applied Sciences*, 11(1).

Rosencrantz, M., Gordon, G., and Thrun, S. (2003). Locating moving entities in indoor environments with teams of mobile robots. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, page 233–240, New York, NY, USA. Association for Computing Machinery.

Rubenstein, M., Cornejo, A., and Nagpal, R. (2014). Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799.

Ruetten, L., Regis, P. A., Feil-Seifer, D., and Sengupta, S. (2020). Area-optimized uav swarm network for search and rescue operations. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0613–0618.

Russel, S. J. and Norvig, P. (2014). *Artificial Intelligence: A Modern Approach, 3rd Edition*, pages 373–407. Person New Int. Edition.

Russell, S. and Norvig, P. (2015). Artificial Intelligence: A Modern Approach. *Available at `http://aima.eecs.berkeley.edu/slides-pdf/chapter05-6pp.pdf`, accessed on 2021-04-21.*

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Art. intelligence*, 5(2):115–135.

Şahin, E., Girgin, S., Bayindir, L., and Turgut, A. E. (2008). *Swarm Robotics*, pages 87–100. Springer Berlin Heidelberg, Berlin, Heidelberg.

Sampedro, C., Bavle, H., Sanchez-Lopez, J. L., Fernández, R. A. S., Rodríguez-Ramos, A., Molina, M., and Campoy, P. (2016). A flexible and dynamic mission planning architecture for uav swarm coordination. In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 355–363.

San-Miguel-Ayanz, J. and Ravail, N. (2005). Active fire detection for fire emergency management: Potential and limitations for the operational use of remote sensing. *Natural Hazards*, 35(3):361–376.

Saska, M., Vonásek, V., Chudoba, J., Thomas, J., Loianno, G., and Kumar, V. (2016). Swarm distribution and deployment for cooperative surveillance by micro-aerial vehicles. *Journal of Intelligent & Robotic Systems*, 84(1):469–492.

Sastry, N. (2002). Forest fires, air pollution, and mortality in southeast asia. *Demography*, 39(1):1–23.

Scherer, J., Yahyanejad, S., Hayat, S., Yanmaz, E., Andre, T., Khan, A., Vukadinovic, V., Bettstetter, C., Hellwagner, H., and Rinner, B. (2015). An autonomous multi-uav system for search and rescue. In *Proc. of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, DroNet '15, pages 33–38, Florence, Italy. ACM.

Schiendorfer, A., Knapp, A., Anders, G., and Reif, W. (2018). Minibrass: soft constraints for minizinc. *Constraints*, 23(4):403–450.

Schiendorfer, A., Steghöfer, J.-P., Knapp, A., Nafz, F., and Reif, W. (2013). Constraint Relationships for Soft Constraints. In *Proc. 33rd SGAI Int. Conf. Innov. Techniques & Applic. of Art. Intelligence (AI'13)*. Springer.

Schmeck, H., Müller-Schloer, C., et al. (2011). *Adaptivity and Self-organisation in Organanic Computing Systems*, pages 5–37. Springer Basel.

Schmickl, T., Thenius, R., Moslinger, C., Timmis, J., Tyrrell, A., Read, M., Hilder, J., Halloy, J., Campo, A., Stefanini, C., Manfredi, L., Orofino, S., Kernbach, S., Dipper, T., and Sutantyo, D. (2011). Cocoro – the self-aware underwater swarm. In *2011 Fifth IEEE Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 120–126.

Schörner, M., Wanninger, C., Hoffmann, A., Kosak, O., Ponsar, H., and Reif, W. (2020). Modeling and execution of coordinated missions in reconfigurable robot ensembles. In *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*, pages 290–293.

Seebach, H., Ortmeier, F., and Reif, W. (2007). Design and construction of organic computing systems. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 4215–4221. IEEE.

Sensirion (2018). Difital Humidity Sensor SHT7x (RH/T). *Available at `https://www.sensirion.com/en/environmental-sensors/humidity-sensors/pintype-digital-humidity-sensors/`, accessed on 2018-11-15.*

Sensornet (2018a). Distributed Temperature Sensing Systems & DTS Monitoring Sensors. *Available at `https://www.sensornet.co.uk/distributed-temperature-sensing/`, accessed on 2018-11-15.*

Sensornet (2018b). Oryx DTS Sensors. *Available at `https://www.sensornet.co.uk/oryx-dts-sensors/`, accessed on 2018-11-15.*

Shehory, O. and Kraus, S. (1998). Methods for task allocation via agent coalition formation. *Art. Intelligence*, 101(1):165 – 200.

Shields, C. (2012). *The Oxford Handbook of Aristotle*. Oxford University Press.

Skrzypecki, S., Tarapata, Z., and Pierzchała, D. (2020). Combined pso methods for uavs swarm modelling and simulation. In Mazal, J., Fagiolini, A., and Vasik, P., editors, *Modelling and Simulation for Autonomous Systems*, pages 11–25, Cham. Springer International Publishing.

Smith, R. G. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. on Computers*, C-29(12):1104–1113.

Smith, S. F. and Cheng, C.-C. (1993). Slack-based heuristics for constraint satisfaction scheduling. In *AAAI*, pages 139–144.

Smys, S. and Ranganathan, G. (2019). Robot assisted sensing control and manufacture in automobile industry. *J ISMAC*, 1(03):180–187.

SNCF Réseau (2015). France: Demonstrating the use of drones for railway maintenance and security. *Available at `https://uic.org/com/enews/nr/464/article/france-demonstrating-the-use-of?page=iframe_enews`, accessed on 2021-01-02.*

Sánchez-García, J., Reina, D., and Toral, S. (2019). A distributed pso-based exploration algorithm for a uav network assisting a disaster scenario. *Future Generation Computer Systems*, 90:129 – 148.

Sousselier, T., Dreo, J., and Brunet, J.-P. (2012). Line formation algorithm in a self-organized swarm of micro-underwater unmanned vehicles. In *2012 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 520–525.

Steghöfer, J.-P., Behrmann, P., Anders, G., Siefert, F., and Reif, W. (2013). Hispada: Self-organising hierarchies for large-scale multi-agent systems. In *Proceedings of the IARIA international conference on autonomic and autonomous systems (ICAS)*. Citeseer.

Stellin, M., Sabino, S., and Grilo, A. (2020). Lorawan networking in mobile scenarios using a wifi mesh of uav gateways. *Electronics*, 9(4).

Stolfi, D. H., Brust, M. R., Danoy, G., and Bouvry, P. (2020). A cooperative coevolutionary approach to maximise surveillance coverage of uav swarms. In *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*, pages 1–6.

Stone, P. and Veloso, M. (2000). Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383.

Sudhakar, S., Vijayakumar, V., Kumar, C. S., Priya, V., Ravi, L., and Subramaniyaswamy, V. (2020). Unmanned aerial vehicle (uav) based forest fire detection and monitoring for reducing false alarms in forest-fires. *Computer Communications*, 149:1–16.

SZ DJI Technology Co., L. (2021). Consumer Drones Comparison. *Available at https://www.dji.com/products/compare-consumer-drones, accessed on 2018-11-15.*

Tahir, A., Böling, J., Haghbayan, M.-H., Toivonen, H. T., and Plosila, J. (2019). Swarms of unmanned aerial vehicles — a survey. *Journal of Industrial Information Integration*, 16:100106.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.

Tate, A. (1976). Using goal structure to direct search in a problem solver. *Informatics thesis and dissertation collection.*

Tate, A. E. and Hendler, J. E. (1994). *Readings in Planning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Thakker, R., Kamat, A., Bharambe, S., Chiddarwar, S., and Bhurchandi, K. M. (2014). Rebis - reconfigurable bipedal snake robot. In *2014 IEEE/RSJ Intern. Conf. on Intel. Robots and Systems*, pages 309–314.

The Gurdian (2020). Hundreds exposed to gas after deadly leak at Indian chemical factory. *Available at https://www.theguardian.com/world/2020/may/07/gas-leak-at-chemical-factory-in-india-kills-hospitalises-lg-polymers, accessed on 2021-02-04.*

Thenius, R., Moser, D., Kernbach, S., Kuksin, I., Kernbach, O., Elena Kuksina, E., Miśković, N., Bogdan, S., Petrović, T., Babić, A., Boyer, F., Lebastard, V., Bazeille, S., Ferrari, G. W., Donati, E., Pelliccia, R., Romano, D., Stefanini, C., Morgantin, M., Campo, A., and Schmickl, T. (2016). subclutron: a learning, self-regulating, self-sustaining underwater society/culture of robots. *Art. Life and Intell. Agents Symposium, 2016.*

Trianni, V. (2008). *Coordinated Motion*, pages 73–95. Springer Berlin Heidelberg, Berlin, Heidelberg.

Trinh, T. T., Trinh, T. T., Le, T. T., Tu, B. M., et al. (2019). Temperature inversion and air pollution relationship, and its effects on human health in hanoi city, vietnam. *Environmental geochemistry and health*, 41(2):929–937.

Tripolitsiotis, A., Prokas, N., Kyritsis, S., Dollas, A., Papaefstathiou, I., and Partsinevelos, P. (2017). Dronesourcing: a modular, expandable multi-sensor uav platform for combined, real-time environmental monitoring. *International Journal of Remote Sensing*, 38(8-10):2757–2770.

Tsang, E. (1993). Chapter 1 - introduction. In Tsang, E., editor, *Foundations of Constraint Satisfaction*, pages 1–30. Academic Press.

Tsang, E. (2014). *Foundations of constraint satisfaction*. BoD–Books on Demand.

University of Augsburg (2016). SASO 2016 - 10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems - University of Augsburg, Augsburg, Germany, 12-16 September 2016. *Available at* `https://saso2016.informatik.uni-augsburg.de/`, *accessed on 2021-05-04*.

Vallejo, D., Castro-Schez, J. J., Glez-Morcillo, C., and Albusac, J. (2020). Multi-agent architecture for information retrieval and intelligent monitoring by uavs in known environments affected by catastrophes. *Engineering Applications of Artificial Intelligence*, 87:103243.

Van der Hoorn, R. A. (2008). Plant proteases: from phenotypes to molecular mechanisms. *Annu. Rev. Plant Biol.*, 59:191–223.

Varughese, J. C., Hornischer, H., Zahadat, P., Thenius, R., Wotawa, F., and Schmickl, T. (2020). A swarm design paradigm unifying swarm behaviors using minimalistic communication. *Bioinspiration & biomimetics*, 15(3):036005.

Vásárhelyi, G., Virágh, C., Somorjai, G., Tarcai, N., Szörényi, T., Nepusz, T., and Vicsek, T. (2014). Outdoor flocking and formation flight with autonomous aerial robots. In *2014 IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, pages 3866–3873.

Véle, A. and Modlinger, R. (2019). Body size of wood ant workers affects their work division. *Sociobiology*, 66(4):614–618.

Vellido, I., Fdez-Olivares, J., and Pérez, R. (2020). A knowledge based process for the generation of htn domains from vgdl video game descriptions. In *ICAPS2020 Workshop on Knowledge Engineering for Planning and Scheduling*, pages 11–20.

VICON (2018). Vicon Object Tracking. *Available at* `https://www.vicon.com/motion-capture/engineering`, *accessed on 2018-11-15*.

Vig, L. and Adams, J. A. (2006). Multi-robot coalition formation. *IEEE Trans. on Robotics*, 22(4):637–649.

Viking (2021). Cl-215, cl-215t and cl-415. *Available at* `https://www.vikingair.com/viking-aircraft/cl-215-cl-215t-and-cl-415`, *accessed on 2021-02-04*.

Villa, T. F., Gonzalez, F., Miljievic, B., Ristovski, Z. D., and Morawska, L. (2016a). An overview of small unmanned aerial vehicles for air quality measurements: Present applications and future prospectives. *Sensors (Basel, Switzerland)*, 16(7):1072–.

Villa, T. F., Salimi, F., Morton, K., Morawska, L., and Gonzalez, F. (2016b). Development and validation of a uav based system for air pollution measurements. *Sensors*, 16(12):2202.

Wan, J., Tang, S., Yan, H., Li, D., Wang, S., and Vasilakos, A. V. (2016). Cloud robotics: Current status and open issues. *IEEE Access*, 4:2797–2807.

Wang, L., Liu, M., and Meng, M. Q. H. (2017a). A hierarchical auction-based mechanism for real-time resource allocation in cloud robotic systems. *IEEE Trans. on Cybernetics*, 47(2):473–484.

Wang, R. H., Sudhama, A., Begum, M., Huq, R., and Mihailidis, A. (2017b). Robots to assist daily activities: views of older adults with alzheimer's disease and their caregivers. *International psychogeriatrics*, 29(1):67–79.

Wang, T., Han, W., Zhang, M., Yao, X., Zhang, L., Peng, X., Li, C., and Dan, X. (2020). Unmanned aerial vehicle-borne sensor system for atmosphere-particulate-matter measurements: Design and experiments. *Sensors*, 20(1):57.

Wanninger, C., Alfano, L., Schörner, M., Hoffmann, A., Kosak, O., and Reif, W. (2021). *under review* - Semantic Plug and Play: An Architecture Combining Linked Data and Reconfigurable Hardware. In *16th IEEE International Conference on Semantic Computing (ICSC)*. IEEE.

Wanninger, C., Eymüller, C., Hoffmann, A., Kosak, O., and Reif, W. (2018). Synthesising Capabilities for Collective Adaptive Systems from Self-Descriptive Hardware Devices - Bridging the Reality Gap. In *8th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation*.

Wedde, H. F. (2012). Dezent – a cyber-physical approach for providing affordable regenerative electric energy in the near future. In *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 241–249.

Weiss, G., editor (2001). *Multiagent Systems*. Messachusetts Institute of Technology, 1 edition.

Weiss, G., editor (2013). *Multiagent Systems*. Messachusetts Institute of Technology, 2 edition.

Weld, D. S. (1999). Recent advances in ai planning. *AI magazine*, 20(2):93.

Willis, C. M., Church, S. M., Guest, C. M., Cook, W. A., McCarthy, N., Bransbury, A. J., Church, M. R., and Church, J. C. (2004). Olfactory detection of human bladder cancer by dogs: proof of principle study. *Bmj*, 329(7468):712.

Wilson, S., Pavlic, T. P., Kumar, G. P., Buffin, A., Pratt, S. C., and Berman, S. (2014). Design of ant-inspired stochastic control policies for collective transport by robotic swarms. *Swarm Intelligence*, 8(4):303–327.

Wolf, B., Chwala, C., Fersch, B., Garvelmann, J., Junkermann, W., Zeeman, M. J., Angerer, A., Adler, B., Beck, C., Brosy, C., Brugger, P., Emeis, S., Dannenmann, M., Roo, F. D., Diaz-Pines, E., Haas, E., Hagen, M., Hajnsek, I., Jacobeit, J., Jagdhuber, T., Kalthoff, N., Kiese, R., Kunstmann, H., Kosak, O., Krieg, R., Malchow, C., Mauder, M., Merz, R., Notarnicola, C., Philipp, A., Reif, W., Reineke, S., Rödiger, T., Ruehr, N., Schäfer, K., Schrön, M., Senatore, A., Shupe, H., Völksch, I., Wanninger, C., Zacharias, S., and Schmid, H. P. (2017). The scalex campaign: Scale-crossing land surface and boundary layer processes in the tereno-prealpine observatory. *Bulletin of the American Meteorological Society*, 98(6):1217–1234.

Yan, Z., Jouandeau, N., and Cherif, A. A. (2013). A survey and analysis of multi-robot coordination. *International Journal of Advanced Robotic Systems*, 10(12):399.

Yang, F., Ji, X., Yang, C., Li, J., and Li, B. (2017). Cooperative search of uav swarm based on improved ant colony algorithm in uncertain environment. In *2017 IEEE International Conference on Unmanned Systems (ICUS)*, pages 231–236.

Yim, M., m. Shen, W., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., and Chirikjian, G. S. (2007). Modular self-reconfigurable robot systems. *IEEE RAM*, 14(1):43–52.

Yuan, C., Liu, Z., and Zhang, Y. (2015). Uav-based forest fire detection and tracking using image processing techniques. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 639–643.

Yuan, C., Zhang, Y., and Liu, Z. (2015). A survey on technologies for automatic forest fire monitoring, detection, and fighting using unmanned aerial vehicles and remote sensing techniques. *Canadian Journal of Forest Research*, 45(7):783–792.

Yuan, Z., Jin, J., Sun, L., Chin, K., and Muntean, G. (2018). Ultra-reliable iot communications with uavs: A swarm use case. *IEEE Communications Magazine*, 56(12):90–96.

Zeeman, M. (2019). ScaleX - scale-crossing intensive research campaigns. *Available at* `https://scalex.imk-ifu.kit.edu`, *accessed on 2021-02-08*. 12.02.04; LK 01.

Zeeman, M. J., Selker, J. S., and Thomas, C. K. (2015). Near-surface motion in the nocturnal, stable boundary layer observed with fibre-optic distributed temperature sensing. *Boundary-Layer Meteorology*, 154(2):189–205.

Zeeman, M. J., Thomas, C. K., Selker, J. S., and Mauder, M. (2014). Recent developments in the use of dts to monitor atmospheric flows. In *AGU Fall Meeting Abstracts*, volume 2014, pages NS41C–05.

Zhang, J., Liu, L., Wang, B., Chen, X., Wang, Q., and Zheng, T. (2012). High speed automatic power line detection and tracking for a uav-based inspection. In *2012 Intern. Conf. on Industrial Control and Electronics Engineering*, pages 266–269.

Zhang, Y., Wang, S., and Ji, G. (2015). A comprehensive survey on particle swarm optimization algorithm and its applications. *Mathematical Problems in Engineering*, 2015.

Zhu, M., Du, X., Zhang, X., Luo, H., and Wang, G. (2019). Multi-uav rapid-assessment task-assignment problem in a post-earthquake scenario. *IEEE access*, 7:74542–74557.

Zlot, R. and Stentz, A. (2006). Market-based multirobot coordination for complex tasks. *The Int. Journ. of Robotics Res.*, 25(1):73–101.

Abbreviations

ABL . . . . . Atmospheric Boundary Layer

$\text{AGG}_{\text{PROTEASE}}$ . . . . . Aggregation Function of PROTEASE

AGL . . . . . Above Ground Level

API . . . . . Application Programming Interface

ASL . . . . . Above Sea Level

$\text{CALC}_{\text{PROTEASE}}$ . . . . . Calculation Function of PROTEASE

CfP . . . . . Call for Proposals

CfRP . . . . . Call for Reconfiguration Proposals

CN . . . . . Complex-Node

CON . . . . . Condition

CSOP . . . . . Constraint Satisfaction (and Optimization) Problem

DE-Fen . . . . . Fendt, Peißenberg, Germany, 47.827600 N, 11.059959 E, Experimental Site

DTS . . . . . Distant Temperature Sensing

GPS . . . . . Global Positioning System

$\text{GROUP}_{\text{PROTEASE}}$ . . . . . Group Function of PROTEASE

HTN . . . . . Hierarchical Task Networks

IMU . . . . . Inertia Measuring Unit

Jadex . . . . . Jadex Active Components Framework

M . . . . . Method

MAPLE . . . . . Multi-Agent Script Programming Language for Ensembles

MARA . . . . . Multi-Agent Resource Allocation

MAS . . . . . Multi-Agent Systems

MRS . . . . . Multi-Robot Systems
MRTA . . . . . Multi-Robot Task-Allocation Problem

NBL . . . . . Nocturnal Boundary Layer
NPC . . . . . Non-Player Characters

OP . . . . . Operator

PDDL . . . . . Planning Domain Definition Language
PM . . . . . Particulate Matter
PN . . . . . Primitive-Node
ppm . . . . . parts per million
PROTEASE . . . . . Algorithmic Pattern for Trajectory-Based Swarm Behavior
PSO . . . . . Particle Swarm Optimization
PWS . . . . . Planning-Time-Worldstate-Modification-Node

RAP . . . . . Resource Allocation Problem
RAPI . . . . . Robotics Application Programming Interface
ROS . . . . . Robot Operating System
RP . . . . . Replanning-Node
RWS . . . . . Runtime-Worldstate-Modification-Node

S&A . . . . . Sensors and Actuators
SAR . . . . . Search and Rescue
ScaleX . . . . . Scale Crossing Intensive Research Campaigns
SCORe . . . . . Search, Continuously Observe, and React
SDH . . . . . Self-Descriptive Hardware
SELF-MADE . . . . . Distributed, Self-Aware Market-Based Mechanism for Ensemble Formation
SN . . . . . Split-Node
SO . . . . . Self-Organization

$\text{TERM}_{\text{PROTEASE}}$ . . . . . Termination Function of PROTEASE
TRANSFORMAS . . . . . Task and Resource Allocation Strategy for Multi-Agent Systems

UAV . . . . . Unmanned Aerial Vehicle
UGV . . . . . Unmanned Ground Vehicle
USV . . . . . Unmanned Surface Vehicle
UUV . . . . . Unmanned Underwater Vehicle

WS . . . . . Worldstate

Nomenclature

α . . . . . an agent

$\mathcal{C}_\alpha$ . . . . . set of capabilities an agent provides

$\mathcal{A}_{\mathcal{E}^\rho}$ . . . . . the set of agents within an ensemble $\mathcal{E}^\rho$

$\mathcal{A}$ . . . . . a not further specified set of agents

ALT . . . . . altitude, a geographic coordinate specifying the elevation above sea level of a point on the Earth' surface

$\mathcal{C}$ . . . . . the set of all capabilities

$\mathcal{C}^p$ . . . . . the set of all physical capabilities

$\mathcal{C}_t$ . . . . . the set of capabilities a task t requires for its execution

$\mathcal{C}^v$ . . . . . the set of all virtual capabilities

c . . . . . a single capability frrom the set of all capabilities $\mathcal{C}$

$c^p_{\text{CARRY-DTS}}$ . . . . . capability for carrying the DTS sensor (i.e., the associated glass fiber cable)

c^p_{EVAC} . . . . . call out an evacuation message at a given position

$c^p_{\text{EXT-FIRE}}$ . . . . . capability for extinguishing a fire

$c^p_{\text{M-PERS}}$ . . . . . capability for measuring a person's presence, i.e., determining whether or not there is a person present at the measuring location

$c^p_{\text{M-DIST-G}}$ . . . . . capability for measuring the distance to the ground

$c^p_{\text{M-DISTRIBUTION}}$ . . . . . capability for estimating the distribution of a certain measure, e.g., a specific gas type

$c^p_{\text{M-DTS}}$ . . . . . capability for measuring temperature using DTS

$c^p_{\text{M-FIRE}}$ . . . . . capability for measuring fire, i.e., determining whether or not there is a fire present at the measuring location

$c^p_{\text{M-GASG}}$ . . . . . capability for measuring the concentration of a specific gas g

$c^p_{\text{M-GASX}}$ . . . . . capability for measuring the concentration of a specific gas x

$c^p_{\text{M-HUM}}$ . . . . . physical capability for measuring humidity

$c^p_{\text{M-PM}}$ . . . . . physical capability for measuring particle matter

$c^p_{\text{M-POS}}$ . . . . . physical capability for measuring a position

$c^p_{\text{M-POS-ID}}$ . . . . . physical capability measuring the position of an identifiable object

$c^p_{\text{M-TEMP}}$ . . . . . physical capability for measuring temperature

$c^p_{\text{M-VEL}}$ . . . . . physical capability for measuring the current velocity

$c^p_{\text{M-WIND}}$ . . . . . physical capability for measuring wind

$c^p_{\text{MV-POS}}$ . . . . . physical capability for moving to a position

$c^v_{\text{MV-SENSOR}}$ . . . . . virtual capability for moving to a position while adapting the height based on the results of a sensor

$c^p_{\text{MV-VEL}}$ . . . . . physical capability for moving with a velocity

c^p_{Υ} . . . . . a physical capability with direct access to a S&A

c^v_{PROTEASE} . . . . . a virtual capability for participating in PROTEASE

$c^v_{\text{TEMP-GRAD}}$ . . . . . virtual capability for moving with a velocity until detecting a different temperature gradient

c^v_{Υ} . . . . . a virtual capability has only indirect access to physical hardware but can combine a set of c^p (indicated with ★) for achieving more complex behavior

★ . . . . . the combination operator for virtual capabilities

$\mathcal{CP}$ . . . . . cooperation pattern for the agent level part of an Ensemble Program

$\mathcal{CI}$ . . . . . information necessary for coordinating control flow and information flow during in a plan

$\mathcal{E}$ . . . . . an ensemble defined as a subsystem of the Multipotent System

$\mathcal{E}^\rho$ . . . . . an ensemble defined as a subsystem of the Multipotent System, formed for a specific plan ρ

EPU . . . . . anensemble processing unit that can be used in an Ensemble Program

EPU . . . . . the set of ensemble processing unit that can be used in an Ensemble Program

c^v_{EXT} . . . . . a virtual capability for participating in an external program

$\rightarrowtail$ . . . . . an *injective* function, mapping each input to a maximum of one output value (or none)

$\mathcal{I}$ . . . . . the Instruction-Set for an Ensemble Program

INSTR . . . . . one instruction from the Instruction-Set consisting of a capability and its functional and non-functional parameters

LAT . . . . . latitude, a geographic coordinate specifying the north-south position of a point on the Earth' surface

LON . . . . . longitude, a geographic coordinate specifying the east-west position of a point on the Earth' surface

$\mathcal{MS}$ . . . . . the Multipotent System consisting of a set of agents $\mathcal{A}_{\mathcal{MS}}$ and a set of self describing hardware $\mathcal{SDH}_{\mathcal{MS}}$, $\mathcal{MS}$:= ($\mathcal{A}_{\mathcal{MS}}$, $\mathcal{SDH}_{\mathcal{MS}}$)

$\mathcal{A}_{\mathcal{MS}}$ . . . . . the set of agents α within the $\mathcal{MS}$

$\mathcal{SDH}_{\mathcal{MS}}$ . . . . . the set of different SDH within the $\mathcal{MS}$

PART-ρ . . . . . a partial plan a HTN in MAPLE consists of

ρ . . . . . a plan derived from a HTN

α^{ρ} . . . . . a planning agent that can be used in a HTN created with MAPLE

$\alpha^{\rho}_{\forall}$ . . . . . planning agent group addressing all other agents named

$\alpha^{\rho}_{\exists}$ . . . . . planning agent group addressing one not further specified other agent named

$\mathcal{A}^{\rho}_{G}$ . . . . . the set of all planning agent groups

$\mathcal{A}^{\rho}$ . . . . . the set of all planning agents that can be used in a HTN created with MAPLE

$\{\alpha^{\rho}_{1}, ..., \alpha^{\rho}_{n}\}$ . . . . . planning agent group addressing a set of specified agents

$\mathcal{A}^{\rho}_{I}$ . . . . . the set of all identified planning agents

$\alpha^{\rho}_{\{\substack{\text{MIN}\\\text{MAX}}\}}$ . . . . . planning agent group addressing a swarm of agents

PC . . . . . a program counter from an Ensemble Program

PFC . . . . . the program flow coordinator of an Ensemble Program

$c^{p}_{\text{REFILLSEEDS}}$ . . . . . capability for refilling a seeding vessel

ζ . . . . . function describing the set partitioning of a $\mathcal{SDH}_{\mathcal{MS}}$

$\mathcal{SDH}^{\mathcal{E}_{\rho}}$ . . . . . the set of SDH within an ensemble $\mathcal{E}^{\rho}$

$\mathcal{SDH}$ . . . . . the set of all SDH within the Multipotent System

$\mathcal{SDH}_{\alpha}$ . . . . . the set of SDH an agent α is configured with

ω . . . . . function determining the total weight of a set of SDH

c^{p}_{SOW} . . . . . capability to sow a seed in the farmwork scenario

t . . . . . a task

$\mathcal{T}$ . . . . . a not further specified set of tasks

$\mathcal{T}^{\rho}$ . . . . . the tasks contained in a plan ρ

WS . . . . . the distributed variables storage