Managing safety and mission completion via collective run-time adaptation

Bozhinoski, Darko; Garlan, David; Malavolta, Ivano; Pelliccione, Patrizio

published in
Journal of Systems Architecture
2019

DOI (link to publisher)
10.1016/j.sysarc.2019.02.018

document version
Publisher's PDF, also known as Version of record

Journal of Systems Architecture 95 (2019) 19–35

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Managing safety and mission completion via collective run-time adaptation

Darko Bozhinoski[a,*], David Garlan[b], Ivano Malavolta[c], Patrizio Pelliccione[d,e]

[a] IRIDIA, Université Libre de Bruxelles, Belgium
[b] Carnegie Mellon University, Pittsburgh, USA
[c] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
[d] Chalmers University of Technology – University of Gothenburg, Gothenburg, Sweden
[e] University of L'Aquila, L'Aquila, Italy

ARTICLE INFO

Keywords:
Collective run-time adaptation
Ensembles
Mission completion
Safety

ABSTRACT

Mobile Multi-Robot Systems (MMRSs) are an emerging class of systems that are composed of a team of robots, various devices (like movable cameras, sensors) which collaborate with each other to accomplish defined missions. Moreover, these systems must operate in dynamic and potentially uncontrollable and unknown environments that might compromise the safety of the system and the completion of the defined mission. A model of the environment describing, e.g., obstacles, no-fly zones, wind and weather conditions might be available, however, the assumption that such a model is both correct and complete is often wrong. In this paper, we describe an approach that supports execution of missions at run time. It addresses collective adaptation problems in a decentralized fashion, and enables the addition of new entities in the system at any time. Moreover, it is based on two adaptation resolution methods: one for (potentially partial) resolution of mission-related issues and one for full resolution of safety-related issues.

1. Introduction

In the near future Mobile Multi-Robot Systems (MMRSs), will be used extensively to perform missions in everyday life and open new business and societal opportunities. MMRSs are an emerging class of systems that can adapt their behavior at run-time to achieve specific goals. They are represented by a set of mobile robots operating as a team with other agents (the term "agents" refers here to generic entities like cameras, ground stations, or even humans) in a shared environment. MMRSs should be able to operate under dynamic, uncontrollable and partially or fully unknown environments. That introduces a set of uncertainties resulting from incomplete knowledge of the run-time structure of a MMRS (e.g., number of agents performing a particular task at a specific moment) and the environment in which the MMRS operates. Incomplete knowledge of the run-time structure of MMRS comes from its openness. By "openness" we mean that new entities can join or leave the system at run time. Incomplete knowledge of the environment comes from its dynamics and uncontrollability (e.g., a bird flying in the environment). The consideration of the environment when specifying the system arises from the fact that a mission is always associated with a physical context within which it is happening, so how a system will perform a mission strongly depends on the environment where it operates (e.g., the system will operate differently in environments with smooth vs. rough surface, environments with many static obstacles vs. environments with a small number of obstacles).

Handling uncertainty up front is often unfeasible (or expensive). This implies that we need to deal with it when the knowledge becomes available at run time. The construction of MMRSs is significantly more challenging than traditional systems due to their mission-criticality (meaning a loss of resources can lead to possible reduction in mission effectiveness) and their safety-criticality (meaning a failure or defective design could cause risk to human life and the environment). Commercialisation and adoption of MMRSs in dynamic environments will only occur if safety aspects are considered and incorporated as first class concerns in the design of the system. Certification bodies should assure some type of safety certification that relies on a complete understanding of the system. However, for mobile robots that operate in dynamic environments it is quite challenging to consider all variants of the overall system due to their adaptive behaviour. Hence, having the ability to analyse and reason about safety independently from the mission requires a clear separation of concerns between safety and mission related issues.

System designers should be able to precisely specify different adaptation solutions with specific guarantees for the different agents in MMRSs in order to ensure high operational confidence. Moreover, they should be able to craft different adaptation solutions that can be reused across missions, projects, and organizations to minimize development cost.

* Corresponding author.
E-mail addresses: darko.bozhinoski@ulb.ac.be, darko.bozhinoski@gssi.infn.it (D. Bozhinoski), garlan@cs.cmu.edu (D. Garlan), i.malavolta@vu.nl (I. Malavolta), patrizio.pelliccione@gu.se (P. Pelliccione).

https://doi.org/10.1016/j.sysarc.2019.02.018
Received 15 November 2017; Received in revised form 17 February 2019; Accepted 21 February 2019
Available online 22 February 2019

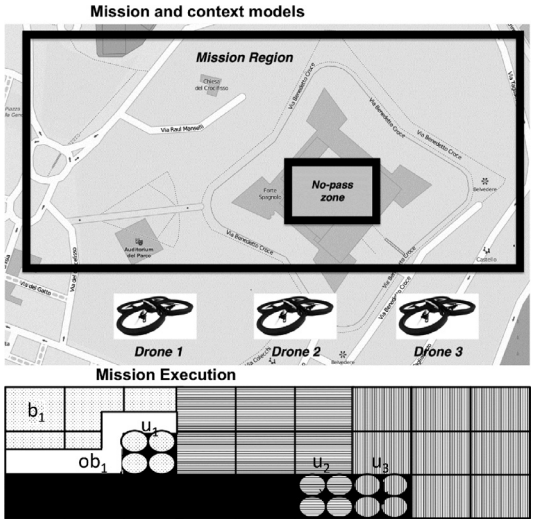

Fig. 1. Motivating scenario.

However, researchers and practitioners have struggled with the lack of approaches to perform different adaptation strategies.

State of the art approaches that support *execution of missions* in MMRSs enable description of the system behaviour under the assumption that all the knowledge used to adapt a system is fully specified at design time and is centrally controlled by a specific component (e.g., [1,2]). Furthermore, most of the proposed solutions do not consider safety aspects separately from the functional behaviour of the robots making the safety certification process more complex and difficult. With our work, we address the aforementioned challenges by proposing a generic approach that:

1) supports *execution of missions* in MMRSs by providing description of the run-time behaviour of different agents in the system.
2) makes a clear separation between mission-related and safety-related adaptation mechanisms. In our work, the system should always satisfy all safety invariants, while the mission can be partially satisfied. Thus, we present two adaptation resolution methods: one for (potentially partial) resolution of mission problems and one for full resolution of safety problems.
3) ensures guarantees about the behaviour of the system.

The rest of the paper is organized as follows. Section 2 describes our motivating scenario of a carbon dioxide monitoring system on which we base our approach. Section 3 describes the framework for mission execution in details, while Section 3.1 discusses the agent's modular reusable behaviours. Section 4 defines a formal model for the problem resolution process. Section 5 presents our mission problem resolution process for (potentially partial) satisfaction of mission objectives, while Section 6 shows our safety problem resolution process for full satisfaction of safety objectives. Finally, related work is examined in Section 7.

2. Motivating scenario

In this section we describe a motivating scenario of a carbon dioxide monitoring system (Fig. 1), which will be used as a running example to explain details of the approach. Fig. 1 shows a single instance of a mission in which three drones have to monitor the CO2 levels within a geographical area; the team of drones has to sense the CO2 level of each geographical point in a grid composed of cells of size 10×10 meters. The upper part of Fig. 1 represents the various mission and context models, where the different colours in the red rectangle represent the different levels of CO2 in the environment (red is an area that has C02 over the treeshold, violet is an area of region that has normal CO2 level, while yellow represents a warning level of CO2 concentration in the environment that is close to the treeshold). This single instance of a mission is considered as successfully completed if the whole area has been fully monitored. Starting from this very high-level description of the mission, the configurations and flight plans for the drones can be automatically generated using tools like FLYAQ [3,4]. Once configured, these drones perform the mission by flying from their initial position to the border of the monitoring area. Then, each drone starts monitoring a specific sub-area so that the whole team can cover the entire area in parallel. A sub-area can be decomposed in number of blocks. A block is the smallest measurable unit for the mission region. Each block is assigned a unique identifier. We represent the size of an area using the number of blocks along each dimension.

The bottom side of Fig. 1 shows the mission execution of three drones. Each drone is executing its corresponding behaviour to cover its part of the mission. The black region is the region that has been already monitored by the corresponding drones. The region with dots represents the region that should be monitored by u_1, the region with horizontal stripes is the region that should be monitored by u_2 and the region with vertical stripes is the region that should be monitored by u_3.

Let us assume that the drone u_1 in Fig. 1 (part of the team U) must reach a target geographical position b_1 and it identifies an obstacle along its trajectory towards the block b_1; if the obstacle is avoidable (e.g., a tree), then u_1 adapts its trajectory to avoid the obstacle to reach b_1; if the obstacle cannot be easily avoided (e.g., the large white object ob_1 in Fig. 1), then the behaviours of u_1 and some other drones in U are adapted so that the block b_1 is covered by another drone $u_i \in U$ and u_1 can cover some other points within the area.

3. Framework for mission execution

Missions are specified at design-time by in-the-field operators. The operators are users that are non-expert in Information and communications technology (ICT), but have specific expertise in the mission domain (like fire fighters, policemen, etc.). Within the panorama of missions [5], mission scenarios might concern: (i) Disaster Prevention and Management, like damage assessment after earthquakes, searching for survivors after airplane accidents and disasters; (ii) Homeland Security, such as coastal surveillance, securing large public events; (iii) Protection of Critical Infrastructure, such as monitoring oil and gas pipelines, protecting maritime transportation from piracy, observing traffic flows; (iv) Communications, like broadband communication, telecommunication relays; (v) Environmental Protection, such as pollution emission, protection of water resources etc. This spectrum of mission requires MMRSs and often they are both mission-critical and safety-critical systems. The definition of missions at design-time include only the information that is available at that time.

A proper management of the run-time phase is required since environments in which these systems have to operate are often unpredictable and unknown. A new plan for adaptation should be computed on-the-fly every time something unexpected in the system or in the environment is observed. However, there are few major challenges in performing adaptation on-the-fly, i.e., computing agents' behaviours on-the-fly. One challenge is to deal with the question of which part of the system should be engaged in an adaptation. This is not trivial at all, since solutions for the same problem may be generated at different levels. For instance, an issue of a robot (i.e., a drop of the battery level of a drone below a safety threshold) can be resolved in the scope of its mission, by re-calculating its navigation plan (isolated adaptation), or in the wider scope with the engagement of other robots and supporting systems (e.g., a drone trajectory manager) (collective adaptation). The challenge here is to understand these levels and create a mechanism that decides the right scope for an adaptation for a given issue. The other challenge is to understand how multiple entities in a collective adaptation can adapt altogether and transactionally and what type of negotiation must take

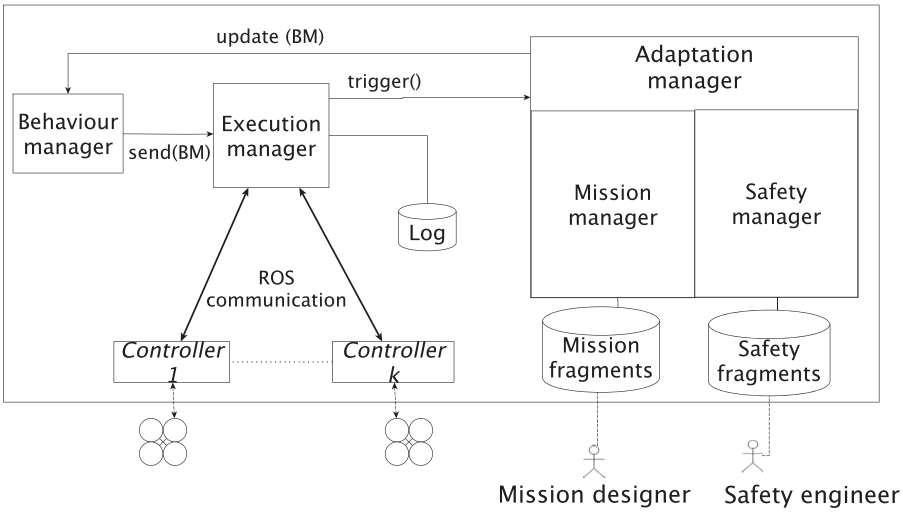

Fig. 2. Overview of our approach.

place to decide the right behavioural changes that should be applied on each side. Moreover, managing the execution of complex missions requires a clear separation of concerns between safety and mission issues. This way an operator can focus on the mission functional specification, while a safety engineer can only focus on safety-specific mechanisms that are generic and independent from the functional behaviour of the system.

In this work, we propose a generic approach for supporting *execution* of MMRSs missions. A possible design-time phase for constructing the behaviour plans of the different agents can be FLYAQ [3,4]. FLYAQ is a platform for the specification of missions of autonomous multicopters through a high-level and graphical domain specific language tailored to the specific application domains. In FLYAQ [3,4] the tool generates behaviour plans at design-time for each of the robots involved in the mission according to the initial set of active robots and the mission specification. The tool associates a robot r_i with a region R for each of the mission tasks t. The correctness of the algorithms employed in the FLYAQ framework regarding preserving safety is proved in [6].

Here we focus on the run-time phase. Most of the existing works are based on the assumptions that the environment is static and that each agent has either a global communication range (can communicate with each other agent in the system) or that each agent can obtain a full knowledge of the system and the environment at any time. These assumptions, however, do not usually hold in real-world scenarios [7]. In our work, an adaptation is performed on-the-fly every time an unexpected system or environmental feature is observed in a part of the system. That being said, a new behavioural plan is computed for one or a group of agents that are affected by it. Our approach supports on-the-fly adaptation that enables MMRSs to complete the defined mission while guaranteeing the preservation of safety constraints. As shown in Fig. 2 the architecture of our approach consists of 3 main components:

- *Behavior Manager:* contains the behaviour model of the mission. Here, behaviour plans are stored as behaviours in a Behaviour Tree [8]. Each agent in the system is assigned a Behaviour Tree. In the beginning of the mission, the Behavior Manager contains all behaviours that should be performed for completion of the mission. During mission execution, the behaviour trees may be updated as a result of violations in the behaviour plans to one or more agents.
- *Execution Manager* is in charge of: (i) receiving the current Behaviour Model (BM) of the mission from the *Behaviour Manager*, (ii) interacting with the controllers both to send their part of mission to be executed and to receive telemetry data, (iii) checking when some conditions in the behaviour plans are violated in order to trigger the *Adaptation Manager*, and (iv) to log mission data.
- *Adaptation Manager* is a component where the adaptation happens. It receives from the *Execution Manager* the conditions that are violated and depending on the type of conditions that are violated it triggers one of its subcomponents. If safety-related conditions are violated the *Safety Manager* is always triggered. The *Safety Manager* is a safety-specific adaptation component that can manage only safety-related problems. If there isn't any violation of safety-related conditions and there are mission-related conditions that are violated, the *Mission Manager* is triggered in order to perform mission problem resolution.

Based on the different type of issues: mission related vs. safety related the approach proposes different adaptation mechanisms which decide the right scope for an adaptation. Safety is a first class concern in our missions as robots can collaborate with humans to accomplish the mission. In this context, the system should always satisfy all safety invariants, while the mission can be partially satisfied. That is why distinguishing between safety-related and mission-related issues is of most importance in our approach. As the nature of the mission objectives is different to the safety objectives, we propose two different adaptation resolution methods: one for partial satisfaction of mission objectives and one for full satisfaction of safety objectives. In this work, a MMRS might need an adaptation due to the following:

- the system cannot successfully complete the defined mission (mission objective);
- agent(s) performing the current mission may physically collide (safety objective).

The *Safety Manager* contains "safety" solvers which are algorithms that generate a behavior for collision avoidance. These are agent-specific and defined independently from mission definition. The *Mission Manager* contains solvers which generate a behavior for completing parts of the mission. These are mission-specific and employed before the start of the mission.

In our approach we took in consideration the following types of uncertainty that the system might face and can be a reason for adaptation:

- *Changing Availability of resources:* The availability of resources for an agent can change over time (e.g., the battery level of a robot is less then a certain value, so the robot cannot finish a task);
- *Change of environment conditions:* The environment where the agents operate is dynamic (e.g., a dynamic obstacle appears, so a robot cannot finish a task).

Even though the agents participating in the mission are autonomous, they are able to dynamically form collaborative groups, called ensembles [9] to gain benefits that otherwise would not be possible. The exam-

ple of such a collaborative group is an ensemble of drones that cooperate in a carbon dioxide monitoring mission represented in the motivating scenario in Fig. 1. Multiple entities must follow certain rules in the ensemble and in return the ensemble offers certain advantages with respect to having single entities working independently. Adherence to these collective rules temporarily reduces the flexibility of collaborating entities, but has huge impact on a particular quality of the system. We can consider what happens if there is a *fault on a drone* and the drone can not continue with its behaviour. In this case, all the tasks that the drone did not manage to complete need to be redistributed to other entities for successful mission completion, while the faulty drone needs to adapt its behaviour plan to safely exit the mission. Another issue we can consider is a *collision between drones*. In that case an immediate and collective reaction by a group of drones is needed for a collision to be avoided. Here, multiple entities must adapt altogether and transactionally to perform a particular collision avoidance protocol. This shows that in MMRSs two levels of adaptation are possible:

- *Isolated adaptation:* change of a single agent's behavior with predefined behavior templates independently from the rest of the system;
- *Collective adaptation:* collective change of the behaviour of a set of agent's teamed up in an ensemble working towards a particular goal.

In this work, we mostly focus on collective adaptation, even though the approach has capabilities to perform isolated adaptation by providing a simple solution to a specific issue.

Finally, our approach is based around the principle of separation of concerns between mission-related vs. safety-related issues.

In the following sections, we will describe how we model the modular reusable behaviours and how agents adapt when facing with mission and safety-critical issues.

3.1. Modeling modular behaviours

In FLYAQ [3,4], at design-time safe behaviour plans are generated for the agents involved in the mission according to the initial set of active agents and the mission specification. Our assumption is that the algorithms used by mission designers at design-time to generate behaviour plans for the agents in our system is correct as in FLYAQ. Now, we focus on the run-time phase. During mission execution, at each point of time the system can follow the state of the mission. We define a mission state as follows.

Definition 1. A *Mission State* is a tuple $MS = (M, C, A, \tau)$, where: M is the mission that should be performed, C is the context under which is performed and A is the set of agents part of the MMRS system performing it at time τ.

Furthermore, we focus on the execution of behaviour plans. For each agent $a_i \in A$ involved in a mission M we generate a behaviour plan BP^i. We define an execution of a behaviour plan for an agent a_i as follows.

Definition 2 (Executing a Behaviour Plan)**.** We define an *execution of a Behaviour Plan* for an agent a_i at time τ as a tuple $EBP^i_\tau = (a_i, BP^i_\tau, S^i_\tau, R_\tau)$ where:

- $a_i \in A$ is an agent;
- BP^i is the behaviour plan performed by agent a_i;
- S^i_τ is the state of the behaviour plan BP^i at time τ;
- R_τ : is the return status of the Behaviour Plan *R, S, F* (the state region of S^i_τ), and can be equal to either Running (R), Success (S), or Failure (F).

The behaviour plan state space for an agent is partitioned in three partitions: *success, failure*, and *running* when an agent a_i is executing a behavior plan BP_i (Fig. 3). The return status of the Behaviour Plan at time τ reports the status of the behaviour plan execution. The states defined in the *success* partition describe that the behaviour plan has been

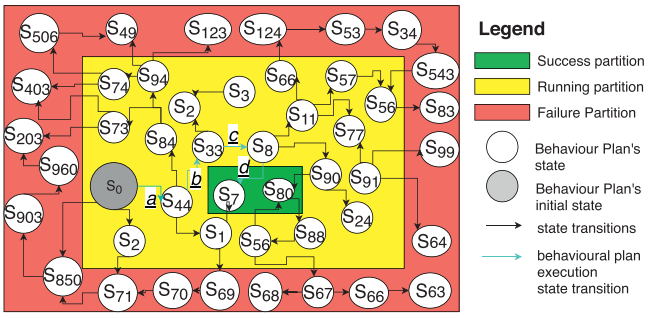

Fig. 3. Partitioning behaviour plan state space.

successfully completed. The states defined in the *running* partition describe that the behaviour plan is correctly executing at time τ. The states defined in the *failure* partition describe that the behavior plan is failing at time τ. In that point the agent should perform adaptation in order to continue executing the mission or just safely exit the mission as described in [4]. In Fig. 3 is represented the behaviour plan state space for an agent a_i executing a behavior plan BP_i. The *behavioural plan execution state transition* $(a \rightarrow b \rightarrow c \rightarrow d)$ represents a behaviour plan state transition from an initial state S_0 to a state in the success region S_7. That is only one possible state transition that could happen. There are many other possible transitions which might include failing into a state from the failure region from where it should adapt.

To model the behaviour plans of an agent a_i we will be using the Behaviour Tree architecture because it provides a flexible mechanism for an agent to switch between different behaviour plans. From Dromey [8], a Behavior Tree is a formal tool that can represent the behaviour of individual entities which change states, make decisions, respond-to/cause events, and interact by exchanging information and/or passing control. It is an organizational execution structure that groups behaviours that one agent should execute as part of its mission. Each behaviour plan of an agent is modeled as a separate behavioural unit. A *Behaviour unit* is one of the basic concepts around which we define our approach. It's an executing structure for explaining what a single agent in the system should do as part of a mission. A behavioural unit is a modular and parametric structure that can be used across missions, projects, and organizations. We believe that modularity is important when designing, testing, and reusing complex mission behaviour in robotics. Individual behaviour units allow individual behavior plans to be easily reused by other robots in other context, without the need to specify how they relate to the whole mission behavior [10].

In the following sections, we will discuss how our approach manages mission and safety issues. We will present an iterative collective adaptation resolution method for partial satisfaction of mission objectives and full satisfaction of safety objectives.

4. Problem resolution model

In this section, we discuss the general problem resolution model which is the base of our approach to manage mission and safety issues.

During normal conditions, each agent performs its behaviour plan generated at design-time and finishes its part of the mission, leading to full mission completion.

However, when an agent executes a behaviour plan and the plan reaches a state in the failure region, it is not correctly executing. When an agent is not "correctly executing" a behaviour plan, a *problem* is triggered.

A problem is a generic structure that corresponds to different critical situations that can happen to an agent when executing a particular behaviour plan. It is generated as a result of the inadequacy between the agent configuration model and the model of its behaviour plan. It can represent situations like a state of an agent that can not cover a partic-

ular mission region because of lack of resources or a state of an agent that represents a situation of possible collision. Each problem includes a set of parameters describing it. We discuss about the problem space we are covering in more details in the next sections. Now, we define a problem formally as follows.

Definition 3 (Problem). A Problem is a tuple $P = (PS, f)$ where

- PS is a generic type of problems;
- f: $PS.p \rightarrow V$ is an assignment function that assigns values $v \in V$ for the set of problems' parameters p. This function defines the boundaries of the problem.

A Solver is a structure that as input receives an instance of a problem and produces a behaviour plan that is a solution to a particular problem. Formally it is represented as:

Definition 4 (Solver). A Solver is a tuple $S = (P_0, PS, SS, \theta)$ where:

- P_0 is the initial problem that should be solved;
- PS is the set of all possible Problems the solver is able to solve;
- SS is the set of all possible solutions;
- θ is a resolution function and $(P_i, B_i) \in \theta$ represents the following:
  - $P_i \in PS$ is the problem that is addressed;
  - $B_i \in SS$ is the Behaviour Plan generated to solve the Problem (solution).

The problem resolution can be performed by one or multiple agents. When multiple agents participate in the mission problem resolution we consider their collective behaviour. Existing approaches typically deal with multi-agent adaptive systems through isolated adaptation: each agent adapts itself independently from each other. However, in our work we consider isolated vs. collective adaptation. Run-time adaptation raises an important issue, i.e., identifying which parts of the system should be engaged in adaptation. This issue is not trivial at all, since a problem may be solved at different scales.

In order to explain our problem resolution process we start introducing the notions of *entity* and *ensemble*. Entities are basic building blocks in the adaptation process representing the different agents of the system (e.g., robots, ground stations, etc.). An entity can be seen as a representation of an agent that can play a *role* in the problem resolution process. Formally, we define it as follows.

Definition 5 (Entity). An entity $y = (a_i, r)$ is defined by an agent a_i playing a role r.

A role represents the type of collaborative interaction a particular agent can participate in. Collaboration consists in managing problems and responding to problems raised. Formally, it is defined as follows.

Definition 6 (Role). A Role is a tuple $R_i = (P, S)$ where:

- P is a set of problems it can produce;
- S is a set of solvers it provides.

The model of an *entity* is primarily determined by the ways it collaborates with other entities as part of an *ensemble*. In isolated adaptation the entity that triggered the problem is the same as the one that provides a solution, but in collective adaptation a solution is provided by other entities that are part of an ensemble. An ensemble is primarily determined by the entities that collaborate to solve a particular problem. In collective adaptation, the ensemble facilitates cooperation between entities by means of an information exchange at run-time. The collaboration between two entities is possible only if the entities can communicate between each other. Formally, an ensemble is defined as follows.

Definition 7. An Ensemble is a dynamic run-time structure represented as a tuple $E = (A, R, \lambda)$ where:

- A is a set of agents grouped together;
- R is a set of roles the agents are playing;

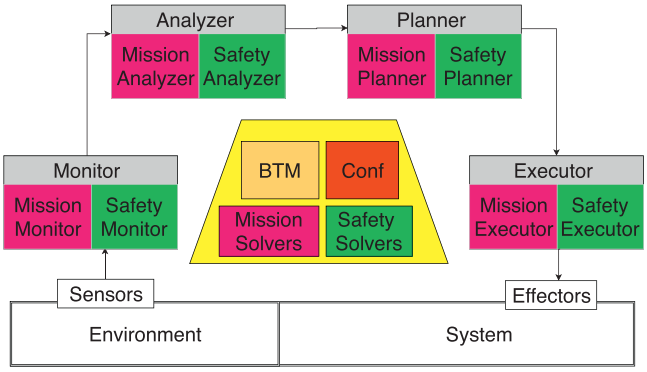

Fig. 4. MAPE-K loop of an entity.

- λ: $A \rightarrow R$ is an assignment function for which the agents are assigned their respective roles (entity definition).

Definition 8. A Problem Resolution is a tuple $PR = (E_i, P_i, S_i)$ where E_i is the ensemble solving a problem P_i and coming with a solution S_i.

To verify the correctness and completeness of our approach using model-checking, we make the assumption that the maximum number of entities that can be part of one ensemble during the problem resolution process is not larger than 16. We consider that this is a reasonable assumption as in practical sense it is difficult to imagine a larger ensemble that can perform complex tasks and give positive outcomes due to the communication overhead and unreliable connection links between entities.

4.1. Representing the MAPE-K loop structure in an entity

In our approach each entity implements the MAPE-K loop [11]. In Fig. 4 is represented a run-time perspective of the entity's MAPE-K architecture. This perspective represents how an entity manages the execution of the mission at run-time while preserving safety constraints. In this section, we will describe each of the MAPE-K loop components for the individual entities. The MAPE-K loop comprises of 4 components operating over a Knowledge base. In order to illustrate the separation of concerns between mission-related and safety-related mechanisms for self-adaptation, there are two sub-components at each stage of the loop, one managing the mission, while the other the safety (Fig. 4). While both sub-components are running in parallel in the Monitor and Analysis to either obtain or update information about the system or the environment, only one subcomponent is running in the Planning and the Executor stage of the loop. In the decision of which component to run, safety has always a precedence over mission completion.

In the *Knowledge* base we define three different types of models that an entity contains. The first model is the Behaviour Tree Model (BTM). The BTM contains all behaviour plans associated with the mission. Each entity has a set of behaviour plans defined at design-time, but only one behaviour plan can be *active* in one point of time during mission execution (depending on its priority). The second model is the current configuration of the entity *Conf*. This model gives information about the current resources of an entity containing information like position of the robot in the map, current level of battery, etc. The third model is a repository of the solvers it can provide i.e. *Mission Solvers* and *Safety Solvers*.

Monitoring component: This component receives stimuli from the environment and from the rest of the system (other entities in the system) and it updates the current configuration *Conf* and *BTM*. Then, it triggers the analysis component. The stimuli are values associated with specific safety-related or mission-related parameters. The *Mission Monitor* keeps track of relevant mission-related information, while the *Safety Monitor* keeps track of safety-related information.

Analysis component: This component makes analysis if the *active* behaviour either *failed, succeeded*, or is *running*. It has two sub-components Mission analyzer and Safety analyzer, both running in parallel and each checking of the appropriate conditions (mission related vs. safety related). Depending on the analysis of the *active* behaviour it does the following:

1) Success: It references the *active* behaviour with the "next" behaviour plan in the behaviour tree model. Then, it triggers the *execution* component;
2) Failed: it triggers the *planning* component;
3) Running: it triggers the *execution* component.

Planning component: This component is triggered when the *active* behaviour returns status failed. The component starts the process of adaptation i.e. which as output generates a behaviour plan that will allow the entity to continue its mission execution or safely exit it. The planning component consists of two subcomponents: *Safety Planner* and *Mission Planner*. When the planning component is triggered, first it gets information from the knowledge about the current configuration model *Conf*. Depending on the type of configuration conditions that are violated it triggers one subcomponent or the other. Safety has precedence over mission completion, so if safety-related conditions are violated the *Safety Planner* is always triggered. The Safety Planner is the safety-specific adaptation component that manages only safety-related problems. If there isn't any violation of safety-related conditions and there are mission-related conditions that are violated, the *Mission Planner* is triggered to perform mission problem resolution.

More details about the problem resolution process will be provided in the next section where we discuss the two problem resolutions: mission problem resolution and safety problem resolution. Each of these resolutions enables two types of adaptation: (i) isolated adaptation: performed by the entity itself or (ii) collective adaptation: performed by an ensemble of entities. The behaviour plan that is generated at the end of the adaptation process is updated in the BTM and then, the execution component is triggered.

Executor component: This component receives the *active* behaviour from the Behaviour Model (BM) and executes (ticks) it i.e. issues commands to the entity's effectors. When the executor component is triggered, it first decides which subcomponent should be activated. Depending on which subcomponent performed the adaptation, it will activate one of the executor subcomponent correspondingly. If the safety planner was activated, the safety executor will be activated. If the mission planner was activated the mission executor will be activated. The mission executor performs mission-related behaviours, while the safety executor performs safety-related behaviours.

5. Mission problem resolution

In an initial work [12], we provided a generic approach for managing run-time adaptation with general types of problems and solvers that can be triggered during mission execution. In this work, we make distinction between mission related vs. safety related problems and solvers. Each entity in the system implements a *Mission Planner* that provides a solution for mission-specific problems. The mission planner receives information about the eligible mission related solvers in the Knowledge base. These are mission-specific and defined before the start of the mission. Mission-specific solvers have a set of solver constraints (configuration parameters) that reduce the space of acceptable problems (e.g., a solver for covering a geographical area might require an entity to have an active camera, enough level of battery etc. to be able to resolve a particular problem). If an entity activates an eligible solver, it generates a solution i.e. a behaviour plan to complete parts of the mission. Here, the scope of mission specific problems is related to the nature of our definition of mission. In order to explain the *mission problem resolution process*, we frame the scope of problems for isolated and collective adaptation.

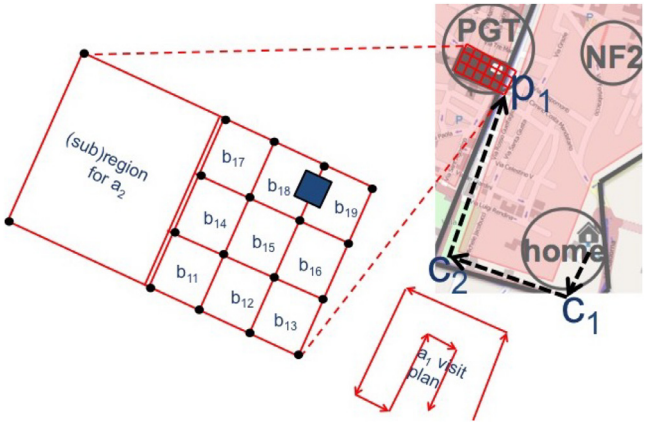

Fig. 5. Task assignment for two agents.

An entity might perform isolated adaptation when facing with a situation where its behaviour plan trajectory needs to pass through a private residence. In this case, the entity might have a solver that generates a behaviour plan for avoiding the region of the private residence. Next, we will speak in more details about collective adaptation.

5.1. Collective adaptation

At design time we assign the mission M to a set of agents A. On Fig. 5 the mission M is assigned to two agents a_1 and a_2. Informally, a region is an area of the mission that is decomposed in a number of blocks (e.g. $b11$, $b12$, $b13$, $b14$, $b15$, $b16$, $b17$, $b18$ and $b19$ in Fig. 5). For each block we assign a unique identifier b_i, $j \in Reg_i$. For each agent $a_i \in A$ involved in the mission M we generate a behaviour plan BP_i covering a region $Reg_i \in M$.

As mission-specific problems strongly depend on the type of tasks the MMRS performs, in this context, we define a mission-specific problem as: a coverage path planning problem for a set of blocks in a region $Reg_i \in M$ that a specific entity $y_i = (a_i, r)$ was assigned to cover, but was not able. The uncovered region $Reg \in M$ is an instance of a *problem*. Accordingly, we focus only on one type of mission-specific solvers which represent *strategies* for covering a region. The mission-specific solvers are formulated as algorithms solving a coverage path planning problem, which depends on the type of the geometry of the mission M. Example of a solver can be an algorithm that generates a solution for covering a region with respect to a specified grid of blocks as in Fig. 5.

As we work particularly with regions, a mission related problem can be decomposed on smaller problems (regions). To be able to annotate the progress of mission execution, we define a measure of satisfiability for a mission M that gives information on the percentage of covered blocks. We indicate that a mission M is completed if all blocks are covered. In contrast, a mission is partially completed if there is a set of blocks $b_{i,j}$ that are not covered.

In collective adaptation, the Mission Manager of an entity can decompose larger problems into smaller ones and can provide a partial solution to the initial problem. The mission manager receives information about the eligible mission related solvers in the Knowledge base and generates a solution i.e. a behaviour plan. The solution is a generated behaviour plan that resolves part of the problem. Our definition of a solver in this particular context allows partial solving of a particular problem due to the fact that problems can be decomposed into smaller ones. In this particular context, we define the size of a problem space through the size (area) of the region that was not covered by the agents.

In this context, the activity of *Mission problem resolution* consists in reducing the problem space of a problem until the problem space is empty or until a timeout occurs. We believe that cooperation in emergent application scenarios requires a new kind of problem resolution approach

```
1  ensemble MissionResolution
2      id ensemble_id: Entity
3      membership
4          roles
5              Leader : Entity
6              Solver_Agent [1..n]: Entity
7      constraints
8          constraint Solver_Agent.hascommpath(Leader)==⊤
9      fitness sum Leader.solution.quality+Solver_Agent.
       solution.quality
10     knowledge exchange
11         Solver_Agent.target = Leader.r_id
12         Solver_Agent.Problem = Leader.Problem
13         Leader.solution=Solver_Agent.solution
```

Fig. 6. EDL specification for mission resolution ensembles.

which is efficient in terms of short delay; so we defined a time deadline until when a solution should be found. If a full solution to the problem is found before the deadline, the mission problem resolution process does not need to wait until the deadline is reached, but it immediately returns the found solution. We formally define a solution of the mission problem resolution as follows.

Definition 9. A solution in the mission problem resolution *PR* is defined as: $Sol = \max_{\emptyset \leq P_i \leq P_o} PR(E_i, P_i, d)$, where E_i is the ensemble solving a problem P_i, P_0 is the initial problem that should be solved, d is a time deadline and *Sol* is the best solution found for that particular time deadline d.

To specify the model of the ensemble needed for problem resolution of mission related problems in details, we will be using the declarative Ensemble Definition Language (EDL) [13]. The main section of the ensemble specification is the *ensemble membership* which defines the structure of the ensemble. A structure of an ensemble is defined through the ensemble roles the agents can participate in. A partial EDL Specification for the mission resolution ensembles is presented on Fig. 6. To identify the ensemble, we declare the id of the entity playing the role Leader to be the *id* of the ensemble essentially saying that instances of this ensemble type cannot be created without being associated with a unique entity instance, which can be seen as a sort of coordinator of the ensemble.

The ensemble membership function consists of three sections. First, the structure of the ensemble is defined by declaring the ensemble roles that the entities can play. In our case, an agent can play one of the following roles in the mission resolution ensemble:

- Leader: an entity that triggers a problem P_k and leads the ensemble formation;
- Solver_Agent: an entity that participate in the solution creation of the aforementioned problem P_k.

An agent can play more then one role in the ensemble i.e. it can be both a leader and a solver_agent (it can trigger a problem, but at a same time it can provide a partial solution to the problem it triggered).

Next, we place semantic constraints, represented by the constraint expression. In our scenario a constraint for an entity to be part of the ensemble specifies that there must be a *communication path* between the corresponding entity and its leader. What we mean by communication path is that if the ensemble leader sends a message in the environment, the information can be transferred from one entity to another, and all entities that are able to receive the information are in the communication path of the leader. In other words, each entity in the ensemble should have a neighborhood region that is overlapping with the neighborhood region of at least one other entity from the same ensemble. Neighborhood *NR* is a region of an entity e_i that gives information with which other entities can communicate in a particular point of time τ. At any point of time during mission execution, each entity in the system has partial view of the system that consists of a list of entities *neighbours* that can communicate with. The third part of the membership definition is the fitness function, specified with a numeric expression. The fitness function provides information about which aspect of the ensemble membership should be optimized. That gives the framework a way to decide which entities should participate in the ensemble formation. More precisely, ensemble instances will be created in such a way to maximize the fitness function. In our example, the fitness function is calculated as a sum of the solution quality provided by all entities in the communication range of the leader. Finally, knowledge exchange is specified, creating an information exchange between the members of the ensemble. In our case, there is an exchange of three types of information: First, the information on the entity that is the leader in the ensemble (line 11), second each of the entities in the ensemble has information about the problem that should be solved (line 12), and third, the solution that the leader obtains for each entity in the ensemble (line 13).

5.2. Mission problem resolution algorithm

We propose a best-effort approach for mission problem resolution, which is efficient in terms of short delay and which does not require knowledge of which and how many entities (agents) are in the system in a particular point of time. To realize our approach, we abstractly define a recursive best-effort algorithm that covers the procedure for mission problem resolution. The algorithm starts from an entity e_i that originally detected a Problem P_i and expects to commit a solution without a time deadline d. Further recursive calls are propagated to other entities in the environment using events.

The algorithm consists of three phases: *discover, construct*, and *commit*.

In the *discover phase*, the possible entities that can participate in the mission problem resolution are found. Each of the entities that can contribute with a solution to a specific problem and have a communication path towards the leader are discovered and links between each of the corresponding entities is created. At any point of time during the mission execution, each entity in the system has partial view of the system that consists of a list of entities *neighbours* that can communicate with. Neighborhood *NR* is a region of an entity e_i that gives information with which other entities can communicate in a particular point of time τ. Each entity that contains an active solver (solver for which the preconditions for activation are fulfilled) can contribute towards a solution to the mission-specific problem if it is in a *communication range* with the entity that triggered the problem. In the *discover phase*, those entities are found and a *temporary* ensemble is formed. The *temporary* ensemble consists of all possible entities that might participate in the resolution process.

In the *construct phase*, each discovered entity in the *temporary* ensemble contributes towards the global solution formation. Each of the entities in the communication range contributes towards the global solution by resolving a particular sub-problem of the general problem, and produces a solution with some specific quality q until a specific deadline d is reached. Hence, solutions are composed from multiple solvers from different entities, the same way problems can be decomposed on multiple sub-problems. In the end, the entity that triggered the problem resolution process decides which is the *best* solution and which part of the temporary ensemble contributes towards it.

In the *commit phase*, the leader of the ensemble knows how well each of the entities can solve a particular sub-problem so it sends request to the entities that can contribute with the best solution to commit their resources. Meanwhile, some of the entities might leave the *temporary* ensemble due to a lack of resources, because they have commitment for another problem resolution process or because they are not anymore in the neighborhood of the entity that triggered the problem. After entities commit their resources for execution, a *stable ensemble* is formed and the *final solution* is obtained. The ensemble that provided the final solution is called *stable ensemble*. It is the ensemble that provides guarantees about the proposed solution i.e. if all behaviours from all entities in the final ensemble execute correctly (according to the generated behaviour plan), the final solution will be guaranteed.

For our algorithm to work we take in consideration the following assumptions:

- All mission problems can be decomposed on sub-problems;
- The entity that triggered the problem does not fail between the time it triggers a problem and commits a solution to an ensemble;
- There exists connectivity between the entity that triggered the problem and at least one other entity in the environment for an adaptation to happen (that entity can be the same entity that triggered the problem);
- The maximum number of entities in one mission problem resolution is 16;
- There is not a noticeable difference in the resource configuration for one entity between the time it proposes a solution and commits its solution;
- There is not a noticeable difference in the connectivity formation between entities that proposed solution and commit their solutions;
- The last two assumptions are translated to the following assumption: there isn't a noticeable difference between the solution quality the moment a solution is proposed and the moment a solution is committed.

The decentralized mission manager for an entity $r_i \in R$ is shown in Fig. 7 in the form of a state machine, which is executed by each entity in the system. It is presented in the form of pseudo-code that closely represents the syntax of the P programming language. A P program comprises of concurrently executing state machines communicating asynchronously with each other using events accompanied by typed data values. Each state machine has an input queue and machine-local store for a collection of variables. Each state has a set of event-handlers, which get executed on receiving the corresponding event. The function *send*(r_k, *ev, d*) is used to send an event *ev* with payload data *d* to target machine r_k. An entity r_i broadcasts event *ev* with payload *d* to all the robots in its communication range using the function *broadcast*(*ev, d*) (more details about the *P* programming language is available at [14]).

Fig. 7 shows the algorithm that encodes the mission-problem resolution state machine. This state machine has three states: *Discover, ConstructSolution* and *RequestCommit*. It contains the following variables which are important for understanding the code: r_i - represents the *id* of the robot that executes the state machine, *P* is the whole problem space for which the entity has already proposed a solution, *S* is the global solution that the leader obtains, *timerV* is a state machine that is instantiated when a problem is triggered, *sol* is a local solution provided by an entity in the ensemble.

The algorithm includes the following important steps:

Lines 2–8. The mission-resolution manager starts executing in the *Discover* state. When a mission related problem is triggered, the solver of the entity r_i is invoked and a solution *sol* is calculated for the specific problem *Pi*. Function *callSolvers* (*line 4*) is beyond the scope of this paper but may generally exploit various mission-specific solvers and provide corresponding full or partial solutions. After the solution is calculated, the machine creates an instance of a Timer machine, starts the timer and goes to *ConstructSolution* state.

Lines 24–43. Upon entering the *ConstructSolution* state, it checks if the solution can fully or partially solve the triggered problem P_i. If we have full solution to the problem P_i, the state machine transits to state *RequestCommit* state (*line 32*). If there is a partial solution, the problem space is reduced to P_j (*line 30*) and the agent broadcasts problem P_j in its neighborhood (communicates the problem P_j with all entities in its communication range).

The events *ReqForSolver* and *SolutionFound* are used for ensemble formation. They create the links between the different ensemble participants that provide solution in the resolution process. For each link, a corresponding problem communication is derived. For each problem communication, a combination of potential solutions is identified across all reachable entities and returned. The entity that triggered the Problem stays in the *ConstructSolution* state until the timeout is reached. When

```
1  machine MissionResolution{
2    start state Discover{
3      on TriggerMissionProblem(P_i) do{
4        sol = callSolvers(P_i);
5        timerV = new Timer(this);
6        StartTimer(timerV);
7        goto ConstructSolution;
8      }
9      on ReqForSolver(P_j, r_j, d) do{
10       if (P_j ∉ P) then
11         sol = callSolver(P_j);
12         if (sol ≠ ∅ ∧ t ≤ d) then
13           P = P ∪ P_j;
14           send(r_j, solutionfound, sol);
15         end if
16       end if
17     }
18     on Commit(sol, r_j){
19       sol=check(sol);
20       update(sol);
21       send(r_j, confirm, (r_j, sol))
22     }
23   }
24   state ConstructSolution{
25     defer ReqForSolver, Commit;
26     entry {
27       if (sol ≠ ∅)
28         Target = r_i;
29         S = sol;
30         P_j = P_i \ p(sol)
31         if (P_j == ∅) then
32           goto RequestCommit;
33         end if
34       else
35         broadcast (ReqForSolver, (P_j, r_i));
36       end if
37     }
38     on SolutionFound(r_j, sol) do{
39       Target = Target ∪ r_j
40       S = S ∪ sol
41     }
42     on TIMEOUT push RequestCommit;
43   }
44   state RequestCommit{
45     entry{
46       < Sbest, Target > = find_best_solution(S,Target)
47       S = ∅;
48       Rrecv = ∅;
49       foreach t ∈ Target
50         send(t, commit, (Sbest(t), r_i))
51       end
52     }
53     on Confirm(r_j, solution){
54       S = S ∪ solution;
55       R_recv = R_recv ∪ r_j;
56       if (sizeof(R_recv) = |Target|) then
57         update(sol);
58         goto Discover;
59       end if
60     }
61   }
62 }
```

Fig. 7. Mission problem resolution algorithm.

the *Timer* machine sends the *TIMEOUT* event to the entity that triggered the problem, the same entity goes to *RequestCommit* state.

Lines 44–62. Upon entering the *RequestCommit* state, the function *find_best_solution* is executed to identify the best solution *sbest* and which combination of entities contributes to *sbest* (*line 46*). The function *find_best_solution* is beyond the scope of this paper but generally is a domain and application specific. Finally, the leader sends the event *commit* to ask the ensemble participants in the specific problem resolution to commit their resources. It should be noted that in the time between the solution is proposed and the solution is chosen, some deviations of resources important for problem resolution might be encountered. That

is why the check function (*line 19*) checks the change in the proposed and the current solution, it updates the Behaviour Tree in the *update()* function (*line 20*) with the current state of the local solution and sends confirmation to the leader. When the leader receives confirmation from all ensemble participants (**line 53**) it updates its Behaviour Tree and transits to *Discover* State. The algorithm provided in Fig. 7 is able to resolve only one problem resolution triggered by one entity at one point of time without any recursion. Here, all the members in the ensemble can directly communicate with the leader. We propose an extension of the algorithm for resolving multiple problems triggered by different entities at a same time. Furthermore, sub-problems are recursively triggered across different entities in the system in order to have a larger range of possible solutions i.e. a leader can obtain a solution from an entity that can not directly communicate with it, but through another entity in the ensemble that is in its communication range. We define a data type $MissionResolution = (PI, Ei, Si, ES, deadline, parent, P0, FE, FS)$, which is a tuple that contains information about one problem resolution (one ensemble) in which the entity participates. It contains the following variables:

- *P0* is the initial problem that is triggered by the entity. It can be sent by another entity that requests collaborators for resolution of a larger problem or it can be generated as a result of the inadequacy between the entity configuration model and the model of its behaviour plan.
- *parent* gives information from where the initial problem originates. If it is generated by the same entity then the parent receives the *id* of the entity r_i.
- *PI* is a reduced version of the problem that is obtained after the entity proposes some solution. We use the value of PI for identifying the different resolution processes in which the entity participates.
- *Ei* stores the information about which are the entities that participate in the temporary ensemble formation.
- *Si* is the solution proposed by the temporary ensemble.
- *ES* is a matrix, which gives information about which entity proposed which solution.
- *deadline* is a timer machine that is instantiated when a problem is triggered and decides until which period of time an ensemble formation is allowed.
- *FE* is the stable ensemble which is obtained after commitment.
- *FS* is the final solution proposed by the stable ensemble.

We can represent each instance of the resolution process as a tree, which we will call problem resolution tree (Fig. 8). On top of the tree, there is an entity that triggered the general problem *P*0, while each node in the lower levels in the tree represents an entity that decomposes the problem of its father entity and provides a partial/full solution. In the end, we have a resolution tree consisting of nodes representing the entities in one possible instance of the problem resolution process. We define a *hierarchical order* in the resolution tree depending on the *communication range* of the entities. The order of an entity in the tree is defined through the hop counter that refers to the number of intermediate entities through which an information must pass between the entity and the leader in the ensemble. For example in Fig. 8, the leader $e1$ might not be able to communicate its problem $P1$ with the entities $e5, \ldots, e10$, which are the leafs in the resolution tree because of the limitation in its communication range, but it might need few entities from the ensemble that are able to transit the information to the leafs (like entities $e2$, $e3$, $e4$), which are in the communication range of the leader e1, but also in the communication range of the leafs in the tree. The order of the entities that can directly communicate with the leader is higher comparing to the entities that need an intermediate entity to relay (in Fig. 8, the leader $e1$ has the highest order, while the leafs $e5, \ldots, e10$ have the lowest order).

A formal definition of the problem resolution tree is as follows.

Definition 10 (Problem Resolution Tree)**.** A problem resolution tree is a tuple $T = (root, Ei, L)$ where:

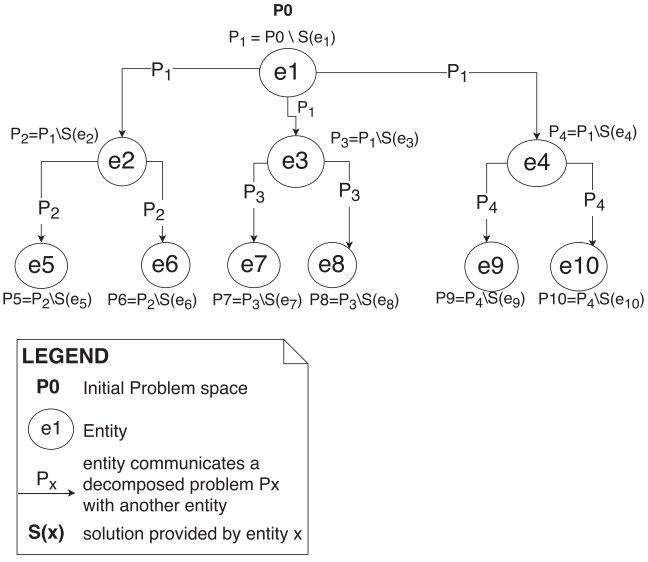

Fig. 8. Problem Resolution Tree.

- *root* is the entity that triggered the top-level problem *P*0.
- E_i are the nodes in the tree represented through the entities that decompose and partially solve part of the top-level problem.
- L: $N \mapsto N$ are parent-child links between entities that are able to communicate between each other. It is a function that represents problems/solutions communications from the root to its children.

Each child in the tree decomposes the problem received from its parent. Then, in the end we have a resolution tree where the leaves are entities that contain the smallest subset of the problem space. Each problem resolution tree represents only one possible instantiation of the problem resolution process. When the problem resolution tree has a full solution, the leafs' problem space is an empty set.

We define a quality q of a solution S_i proposed by an entity in the problem resolution tree based on two factors: (i) *closeness* to the entity that triggered the problem, (ii) intrinsic quality given by the entity. What we mean by closeness to the entity is the following: in one instance of the problem resolution tree, if there is an entity e_k that has a *higher-order in the hierarchy* in the problem resolution tree and can propose a solution s_k to a sub-problem p_k, then we consider that the solution s_k has precedence over solutions that are able to solve the same sub-problem p_k, but are coming from other entities that have lower order in the hierarchy in this instance of the problem resolution process. Because the communication between entities is limited, the algorithm is searching for solutions closer to the entity that triggered the problem p_k and if it finds one, it stops the search for other solutions that are generated from entities that might produce solutions with better intrinsic quality, but are more distant from the leader in the problem resolution tree. Thus, when we speak about hierarchy, we consider hierarchy of entities in terms of the problem resolution tree: the nodes that are closer to the root (meaning they need less number of hops to communicate with the root) have a higher order in the hierarchy comparing to nodes that are lower in the branching. Root has the highest order in the hierarchy, while the leafs have the lowest order in the resolution tree.

5.3. Correctness and completeness

To resolve mission resolution problems we used a gossiping algorithm that aims to disseminate the information about a specific problem and finds a solution. To prove the correctness and completeness of the approach, we need to prove correctness and completeness of the algorithm. In this section, we prove correctness and completeness of our al-

gorithm i.e. we show that the algorithm is aligned with Definition 9 and always provides the best possible solution for a particular deadline.

5.3.1. Correctness

In order to prove that our algorithm is correct, we need to show that the solution computed by the algorithm is correct and is the best solution for a particular deadline.

For a solution to be correct, we assume that the solvers provided by the different entities in the system are correct. Correctness of a solver means that an entity's solver can generate a behaviour (solution) to resolve a particular mission related problem with a quality q_a.

Moreover, we need to show that the solution provided by the algorithm is the best solution for the specific deadline. The leader when calculating the solution for a particular problem, does not have the exact information on how each entity in the ensemble contributes towards the final solution. The only information the leader of the problem resolution process has when making the decision is how each entity that is directly reachable (it can directly communicates with) can help in resolving part of the bigger problem of the leader. All reachable entities might have formed sub-ensembles that contribute towards the final solution, but the leader does not have that information. For example, a leader might be able to communicate with two entities that can provide some solution to the initial problem. Each of those two entities might have formed a temporary sub-ensemble. The two temporary sub-ensembles are on the same hierarchical level and they might have one entity in-common, however they belong to different instances of the resolution process and can be represented with two different problem resolution trees. When the leader decides for the final solution it might consider a combination of both sub-ensembles which create the final solution and might choose a final *stable ensemble* that is a combination of both sub-ensembles. Here the idea is that at each level of the problem resolution tree each node has calculated the "best solution" provided by its leafs. The process is repeating and in the end, the root of the tree should calculate the "best solution". As mentioned before, we represent this solution provided in one instance of the problem resolution tree as in Fig. 8.

To understand if the algorithm correctly calculated the best solution, we should consider not only the fact if the leader correctly calculated the "best solution", but we also need to take in consideration the structure of the ensemble which participated in the specific resolution that provided the "best solution". The structure and state of the ensemble providing the final solution has high impact on the quality of the solution. That being said, we make the following assumptions. First, we assume that there should not be a noticeable difference in the solution quality between the moment a solution is proposed and a solution is chosen. Solution quality will remain the same if there isn't any change in the *entity's resources* and in the *connectivity formation* of the ensemble. That is why we assume that the time between a solution is proposed and a solution is committed is within seconds, so that there isn't any change in the *connectivity formation* of the ensemble (*Assumption 1*). However, change in the resources for the proposed solution might come if an entity participate in two different resolution processes triggered by different leaders. During problem resolution, entities might propose solutions for different problems in different ensembles. As the resources of the entities are limited, we made the assumption that the entity will not participate in two different resolution processes (resolution processes triggered by two different leaders) which overlap in the usage of the resources i.e. if an entity participates in two different resolution processes from different leaders the usage of resources will not overlap (*Assumption 2*). However, there is another case that impacts the quality of the solution and that is when an entity tries to propose multiple solutions in one problem resolution process. In Theorem 1 we show that this would not be possible. Now, we show that our resolution algorithm for solving mission problems is correct by proving that it satisfies the following theorem.

Theorem 1 (Correctness). *If an entity e triggers a problem P_i and Assumption 1 and Assumption 2 are true, then the Mission Problem Resolution Algorithm finds and computes the best quality solution S_i proposed by an Ensemble E that is in the communication range of the entity e.*

Proof. As we mentioned earlier we assume that all mission related problems can be decomposed. Let's say we have an entity e that triggered the problem *Pi*. *Pi* can be decomposed on m different different ways. For each different decomposition there is a sequence of local solutions proposed by an ensemble E_m that combined together give a global solution S_m. The global solution S_m is a sequence of local solutions $s_0, s_1, \dots, s_k$, each of them with a particular quality $q_0, q_q, \dots, q_k$ correspondingly. Let's assume there exists an ensemble consisting of n entities for which there is a communication path between them and the leader (the ensemble might include the leader) and that they can provide the best final solution S_i to a problem P_i. Correspondingly, we can decompose the solution to a sequence of local solutions $S_i = (s_0, s_1, \dots, s_n)$ each with a particular quality $q_0, q_1, \dots, q_n$. We can represent that ensemble using the problem resolution tree (Fig. 8). In order to prove that the algorithm is correct, we need to prove that the computed solution S_i by the leader of the stable ensemble E_n is the best for the problem P_i. To prove that, we use the problem resolution tree of the *stable ensemble*. The root of the tree is the leader. We need to prove that the solution calculated by the leader has the highest quality i.e. $S_i = (s_0, s_1, \dots, s_n)$ each with a particular quality $q_0, q_1, \dots, q_n$. As our mission problem resolution is recursive at each node in the resolution tree, the algorithm calculates the best solution by considering the best combination of solutions proposed by its children. After calculating the best solution, it sends that solution to its parent. Starting from the leafs of the tree, the nodes calculate the best combination of solutions. In the end, the leader composes all combinations and calculates the best combination of solutions proposed by its children. If an entity proposed solutions to multiple problems in the same instance of the problem resolution tree, the algorithm will return a value which might not be correct because of lack of resources for the entities that proposed multiple solutions.

That is why we need to have (i) a set of n *different agents* in the *stable ensemble* contributing towards the final solution S_i as a precondition for the leader to be able to correctly calculate the best possible solution. Our algorithm should not allow for one entity to participate with multiple different solutions in a same ensemble because as we mentioned earlier it might not be possible for one entity to perform multiple solutions due to a lack of resources.

To prove (i), we need to prove that there is no possibility for a *communication loop* in one instance of the problem resolution process (problem resolution tree). What is considered as a possible loop in this distributed algorithm is a situation where one entity communicates a sub-problem $P_j \in P_i$ with another entity that reduce the problem to $P_k \in P_i$ and communicates that problem to the first entity that triggered P_i. We can imagine a situation where before an entity commits its resources to a particular ensemble, it might propose solutions for other ensembles to resolve different problems, so in that case we might encounter a situation where the first entity will propose a solution to a sub-problem that was not able to resolve it before taking in consideration the whole nature of the problem (ex. in a previous iteration the entity proposed a solution to a more general problem and if it commits its resources to that solution, it might not be able to resolve the smaller problem that requires a solution in this iteration). To avoid that, each entity that runs the algorithm checks if the problem that is being received for resolution is some type of sub-problem of a more general problem that was being resolved in a previous iteration. If that is the case, the problem resolution procedure will not start i.e. the entity will not participate in the execution of the sub-problem.

In order to prove the correctness of the algorithm, we verified (using model-checking) the following property: (i) for each problem resolution tree, all nodes in the tree represent different entities in the system (for up to 16 robots). We used Zing model-checker [15] to systematically test our algorithm represented in the state-machine based programming language P. Zing is a model checker used for verification of concurrent

software. Zing explored the state space of our system model starting from the initial state exploring reachable states for up to 16 robots in a depth-first manner. From here we can conclude that for each instance of the resolution procedure for a problem P_i we have a set of $n \leq 16$ *different entities* that contribute towards the solution S_i. In other words, there aren't two nodes in the tree that have reference towards same entity.

Hence, we proved correctness of our algorithm. □

5.3.2. Completeness

Completeness of the approach is defined based on assumptions about the connectivity between the agents and the stability of resources and connectivity. First, we assume that we have complete connectivity between agents meaning that starting from each agent we can broadcast a message that will reach all agents in the environment for a specific deadline d. Second, we assume that we have stable connectivity between agents which means that the connection between the agents will not disrupt during the adaptation process. Third, we assume that we have stable resources during adaptation which means that there isn't a change in the resources important for the agent to execute the solution it proposes. Taking in consideration these assumptions, we prove the completeness of the approach as stated in Theorem 2.

Theorem 2 (Completeness). *If a solution S_i for a specific problem P_i exists and we have a deadline d to find it, the algorithm is able to find it.*

Proof. Let us introduce a metric m that is a number that represents the size of the problem. For example, in our motivating scenario, we consider that mission-specific problems are regions that are uncovered, so in this case as a metric m we represent the area of the uncovered region. With this metric, we want to measure the different degrees (levels) of mission fulfilment [16, § 16.1]. In this context, we represent one instance of the resolution process of problem P_i as a monotonic decreasing sequence m_i ($m_{n+1} \leq m_n, \forall n \in \mathbb{N}$), where each number represents the area of a reduced uncovered region.

Generally, we represent each instance of the mission resolution process of problem P_i (we consider that all mission-specific problems can be decomposed) as a monotonic decreasing sequence where each element in the sequence represents the appropriate reductions of the problem P_i. Obviously, if a sequence is decreasing and is bounded below by a minimum, in some finite time we will reach that minimum. In our case, the minimum would be the problem that corresponds to the final solution S_i, which in ideal situation will be the empty set.

If we represent one instance of the resolution process of problem P_i as function that reduces problem P_i, then we can represent each reduction of P_i in a monotonic decreasing sequence where each instance represents the appropriate reductions of the problem P_i. Obviously, if a sequence is decreasing and is bounded below by a minimum, in some finite time we will reach that minimum. In our case, the minimum would be the problem that corresponds to the final solution S_i, which in ideal situation will be the empty set. Having that for all instances of the mission resolution process (which is always a finite number), we can always find a solution S_i. □

6. Safety problem resolution

In this section, we discuss about the adaptation resolution problem related to safety problems. To be able to model specific safety related behaviours, we discuss few properties of agents and how they are connected to safety.

With A we denote the set of all agents performing the mission and with T the total mission execution time.

We take a snapshot of the mission at a particular time $\tau \in T$. We denote with OBS_τ the region of all obstacles (known and unknown) in the environment at time τ. We define an Obstacle $o \in OBS_\tau$ as a region in the environment that should not intersect with the region of operation of an active robot (agent) to not jeopardize safety. We denote by $VR^i_\tau \in R$ the "visible region" in which an agent a_i can "observe" its local environment (obstacles and other agent's location) at time τ and by SR^i_τ the *safety region* of an agent a_i at time τ that represents a region that is the absolute minimum separation for safety that must be maintained during a close encounter with other (robots) agents or with a static/dynamic obstacle. We identify each agent a_i through its safety index. The safety index is a unique identifier that specifies how well an agent can resolve safety issues. Agents that have a higher index have a higher level of safety resolution capabilities. In this work, we focus on one representation of safety defined through the concept of *collision*. We define *collision* as a situation when the safety zone of a robot is overlapping a region of an object or a safety zone of another robot at time τ. In that context, we say that a MMR system is *safe*, if no collision happens during mission execution. Taking that in consideration, we formally specify what a safe MMRS is.

Definition 11 (Safety). We say a MMRS system is *safe* if and only if the following two safety invariants are valid:

1. $\forall a_i, a_j \in A \& \forall \tau \in T; SR^i_\tau \cap SR^j_\tau = \emptyset$.
2. $\forall a_i \in A \& \forall o \in OBS_\tau; SR^i_\tau \cap o = \emptyset$.

We defined safety in Definition 11 in terms of absence of collisions, where (1) states that there will be no collision between agents and (2) states that there will be no collision between agents and obstacles during mission execution.

Our framework implements a safety planner as part of the Planning component in the MAPE-K loop for each entity. The knowledge base of the MAPE-K loop contains a catalogue of correct obstacle avoidance algorithms as solvers which can be activated and able to provide a solution for an agent in a specific situation. These are solvers that can be reused in different application scenarios and missions independently of the type of the domain. Here, the scope of safety specific problems is independent of the nature of the definition of the mission.

In order to explain the safety problem resolution process, we frame the scope of safety problems for isolated and collective adaptation. In our work, we envision resolution of the following types of safety problems: (i) collision with a static obstacle; (ii) collision with a dynamic obstacle; and (iii) collision between agents. In the case of static/dynamic obstacle avoidance, an entity performs isolated adaptation i.e. one entity generates a solution (behaviour plan) to the problem. The generated solution (behaviour plan) brings the entity into a state from which it can continue executing the mission. In the reminder of this section, Section 6.1 describes more in details collective adaptation, and Section 6.2 discusses the correctness and completeness of the proposed algorithms.

6.1. Collective adaptation

For the third type of safety problems (collision avoidance between agents), our framework provides a palette of coordination protocols for the agents to be able to perform collision avoidance maneuvers. To resolve safety problems related to collision between agents, our framework uses the concept of ensembles described above. Agents dynamically form collaborative groups using attribute-based communication ensembles as described in [17] to gain benefits that otherwise would not be possible. In safety problem resolution, the agents must follow certain rules and in return the ensemble offers a guarantee that if all single agents follow the rules, safety will be preserved. Adherence to these collective rules temporarily reduces the flexibility of the collaborating agents, but has a strongly positive impact on safety. Comparing to mission resolution where those collective rules are more flexible, safety resolution requires stronger, more precise, and detailed rules.

A safety ensemble is primary determined by the agents that collaborate to solve a safety problem. In a collision between multiple agents, our safety resolution procedure consists of: (i) a protocol for on-the-fly ensemble formation for safety resolution and (ii) a recursive function

```
1  ensemble SafetyResolution
2      id ensemble_id: Entity
3      membership
4          roles
5              Leader : Entity
6              Solver_Agent [1..n]: Entity
7      constraints
8          constraint Solver_Agent.hascommpath(Leader)==⊤
9          constraint ∃ Solver_Agent.solver
10             compatible(Solver_Agent.solver, Leader.solver)==⊤
11     fitness Leader.solution.quality
12     knowledge exchange
13         Leader.Problem=Solver_Agent.Problem(conflict_r)
14         Solver_Agent.solver = Leader.solver(A, attr)
15         Leader.solution=Solver_Agent.solution
```

Fig. 9. EDL Specification for safety resolution ensembles.

that is initially called locally by the ensemble leader to select and commit a solution. In the safety problem resolution, all entities in the ensemble must participate in the solution creation because full solution is required. What we mean here is that we treat safety as binary (the MMRS is safe or not). In contrast, in the mission resolution a partial solution is enough for solving a particular problem. If one agent fails to comply to the rules in the ensemble, safety will be compromised. Here, the shape and structure of the ensemble is strongly correlated with the type of the safety problem due to the fact that all involved participants need to generate their corresponding behaviours to guarantee the safety of the system.

We specify the ensemble type used for safety problem resolution in Fig. 9. To identify the ensemble, we declare a leader agent to be the *id* of the ensemble - essentially saying that instances of this ensemble type cannot be created without the ensemble being associated with a unique entity instance, which can be seen as a coordinator of the ensemble. A leader of an ensemble for safety resolution is an agent that leads the ensemble formation and decides for a safety resolution protocol. Example could be a fixed camera positioned in a particular point in the environment that checks if there is a possibility for collision between two or more robots or a robot that notices another robot in its visible region. Unlike the leader in the mission resolution ensemble, the leader in a safety resolution has knowledge of all the possible ensemble participants when it decides for a solution type and when it starts the coordination of the problem resolution process.

As we can see from Fig. 9, the ensemble membership function consists of three sections.

First, the structure of the ensemble is defined by declaring the ensemble roles the agents can play. In our case, same as in the mission resolution, an agent can play one of the following roles in the safety resolution ensemble:

- Leader - an agent that has the highest safety index and leads the ensemble formation;
- Solver_Agent - an agent that can provide partial solution that contributes towards the final solution.

Second, we place semantic constraints, represented by the constraint expression. In our safety resolution ensemble a constraint for an agent to be part of the ensemble is that there must be a communication link between the corresponding agent and its leader. What we mean here is that the leader can communicate with all ensemble participants. The other very important aspect in the ensemble formation is the solution space. In contrast to the mission resolution ensemble, a safety resolution ensemble must provide a full solution, so we put that as a constraint. Full solution consists of combination of behaviours generated by all participants in the ensemble. What we mean is that all ensemble participants agree to follow a specific protocol suggested by the leader i.e. each entity in the ensemble must have a solver compatible to the solver proposed by the leader. Our assumption is that each entity has at least one solver that is compatible to the solvers of the rest of the system. We consider this assumption reasonable because we consider safety independently from the mission, so all safety solvers can be independently embedded in the knowledge base before the start of the mission independently of their type.

The third part of the membership definition is the fitness function, providing information about which aspect of the ensemble membership should be optimized. In our example, the fitness function is represented as maximized solution quality of the leader that coordinates all ensemble participants. Finally, knowledge exchange is specified, creating an information exchange between the members of the ensemble. In our case, we have exchange of three types of information. First, it is the information of the agents' conflict region $conflict\_r$, which is calculated by the position of the agent and its corresponding speed (line 13). $conflict\_r$ is the safety region of an agent in a specific time interval during the mission execution. Second, each of the entities in the ensemble receives an information about a specific solver proposed by the Leader. Each entity in the ensemble receives information about a collision avoidance algorithm *A* and the attributes of the algorithm *attr* (example of attributes of the algorithm might be the central point around which the entities will perform the collision avoidance protocol, their corresponding speed, etc.) (line 14). Third, the leader gets information about the solution of each entity in the ensemble (line 15).

We defined the following protocol (Fig. 10) that is used for on-the-fly ensemble formation when agents discover that they are facing with a possible collision among them. Each agent starts executing in the *Discover* state. If an agent a_i notices other agents in its visible region, it broadcasts the *ReqforSafetyRegion* event with identifier for the *VR* - visible region, the time τ when the message is sent and the identifier of the agent *id*, asking for the safety region *SR* of the robots in its "visible region" during some time period $\Delta\tau$. Then it goes to the *WaitForResponse* state.

Depending of state it is, when an agent a_i receives the event *ReqforSafetyRegion(msg)*, it generates a message $m = (a_i : int; conflict\_region : R; ensemble\_state : int)$, where a_i is the identifier of the agent creating the message, $conflict\_region$ is a region in the environment that should not intersect with the region of operation of an active agent. If the agent is not part of an ensemble, the conflict region is equal to the agent's $SR^i_{\Delta\tau}$, while if the agent is part of an ensemble, it represents the *execution region of the ensemble*, which is the union of the $SR^i_{\Delta\tau}$ of all ensemble constituents. The $ensemble\_state$ gives information if the agent is part of an ensemble and if it is, it gives information in which phase of operation the ensemble is. It can be in one of the following states:

- *NO_ENSEMBLE:* means that the agent is not part of an ensemble;
- *INITIALIZATION:* means that the agent is a part of an ensemble that is in the phase of formation;
- *PLANNING & EXECUTION:* means that the agent is a part of an ensemble that is established and it executes some safety-related algorithm that can not be interrupted at that time i.e. at the time of execution.

In the *WaitforResponse* state, the agent a_i waits to receive feedback from all the agents that were found in its visible region *VR*. When it receives message from all the agents in its visible region it goes to a state *Decide*. In the state *Decide*, the agent goes through all messages and for each message it does the following: first, it checks if the agent a_j that sends the message might collide with a_i. If there is a possible collision, it categorizes the message in one of the following categories:

1) *Category 1:* The message is from an agent a_j that is a part of an ensemble in an initialization state (ensemble_state=initialization). In this case, the agent a_i considers the possibility to join to that ensemble.
2) *Category 2:* The message is from an agent a_j that is not part of an ensemble (ensemble_state=no_ensemble). Here, the agent a_i considers the possibility to start with initialization of an ensemble.
3) *Category 3:* The message is from an agent a_j that is a part of an ensemble that is in planning&execution state (ensemble_state=planning&execution). In this case, the agent

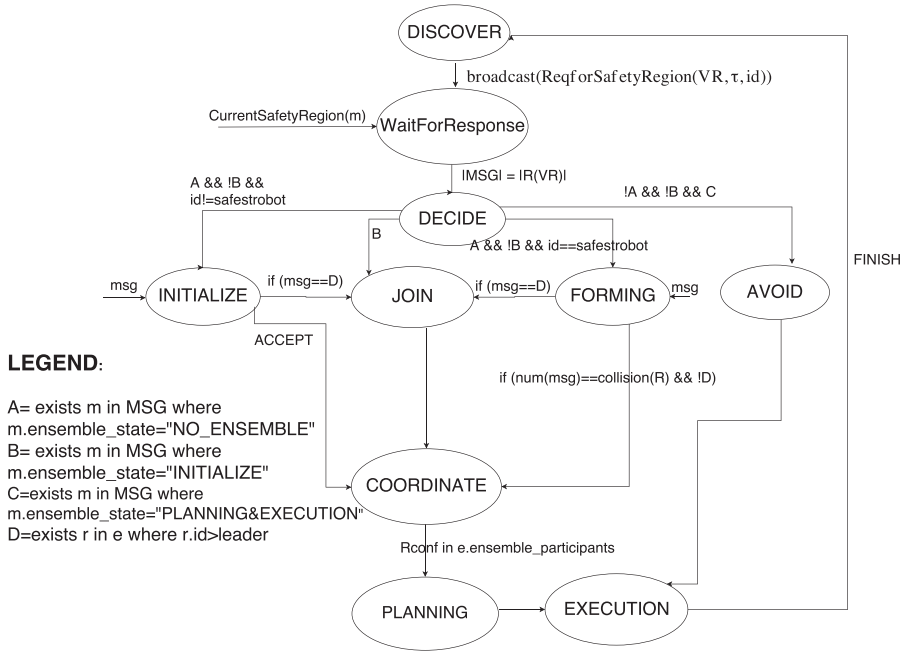

Fig. 10. State machine animating each entity in the safety resolution process.

a_i considers the conflict_region received in the message as a dynamic obstacle and adds it in its collision region *CollisionR*. The ensemble in planning&execution state cannot be interrupted.

After a_i finishes the iteration through all the messages, it does the following decision:

- if there is at least one message from *category 1* (statement B in Fig. 10 is true), the agent a_i goes to state *JOIN* and initiates the joining process. If there are multiple messages from *category 1*, the agent a_i considers joining to the ensemble that has leader with highest safety index.
- if there are no messages from *category 1* (statement B in Fig. 10 is false), it checks if there is at least one message from *category 2*, i.e. if statement A in Fig. 10 is true. If there are multiple messages of that kind, with the function *findsafestrobot* it finds the identifier of the agent with the highest safety index in its visible region. If the agent has the highest safety index in its visible region, it goes into state *FORMING* from where it starts the initialization of the ensemble, otherwise it goes into state *INITIALIZE*.
- This is the case when the agent has received only messages from agents that are parts of ensembles that are in their planning&execution phase (*category 3*). In this case the agent a_i goes into the *AVOID* state where the agent acts as it discovered a dynamic obstacle. In this state the agent should activate some of its solvers for dynamic obstacle avoidance that are able to generate a behavior (solution) for a dynamic obstacle avoidance. Here, the agent performs isolated adaptation and threats this problem on the same level as a dynamic obstacle.

We define conflict set *Rconf* for an entity *e* as the set of entities that are in its visible region, can collide with *e* and are not part of an ensemble that is in planning&execution phase (entities that have send messages from category 2 and category 3). This means that *Rconf* entities are "open" for performing an appropriate collision avoidance protocol that should avoid the possible collision.

From the *INITIALIZE* state, the entity waits for a message *m* from the agent that has the highest safety index in its visible region to decide in which state it should go. If it receives a message that there exists another entity with a higher safety index (statement D is true), it transits to a *JOIN* state from where it starts the procedure for joining an existing ensemble. Otherwise, if it receives an *ACCEPT* message means that the agent with the highest safety index in its visible region started the process for ensemble creation, and then it goes to a state *Coordinate*. That means that there isn't any other entity with higher safety index in both of their visible regions.

The entity in the *FORMING* state is a possible leader (coordinator) of an ensemble. The entity in this state sends ensemble "proposal" requests to the entities that are in its vision region and have safety regions that overlaps with its safety region at some point of time in the future. The entity e_i stays in the *FORMING* state until one of the following happens:

- it receives a message from an entity in its visible region that "rejects" its ensemble "proposal" meaning that the entity that send the message is aware of another entity that has a higher safety index, so that entity should be the coordinator/leader of the ensemble (statement *D* is true). In that case, the entity that received the message directly goes into state *JOIN*.
- it receives a message from all entities in *Rconf* and they all accepted the proposal for ensemble creation. That means statement *D* is false i.e. there isn't any entity in *Rconf* that is aware of another entity e_j that is in the *FORMING* state and has a higher safety index.

In the *Coordinate* state, the coordination between the different entities in the ensemble happens. The entity stays in this state, until all entities represented with *Rconf* are part of the ensemble. When each of *Rconf* entities join the ensemble, the entity goes into a *Planning* state. The last entity that enters the *Planning* state is the leader. When the leader enters this state, on-the-fly ensemble formation phase finishes and the final ensemble is formed. Here, the leader has information for all ensemble participants and the final conflict region of the ensemble. In this state the leader makes a decision of what is the best solver i.e. what is the best collision avoidance protocol that all ensemble participants should follow. We implemented that by using the function *coordinate*() which is executed only by the ensemble leader (Fig. 11). Based on the information about the ensemble (conflict regions, participants, etc.), the leader generates a prioritized list of all possible solvers (list of collision avoidance algorithms) embedded in the Knowledge base that could solve the possible collision for this ensemble formation (line 3). Then, the ensemble leader starts from the first algorithm and calls the recursive function $resolve\_safety(A, id, e\_j)$ where *A* is the algorithm that is

```
1  function coordinate()
2      if (id == ensemble.leader)
3          SolverList = callSolvers(ensemble);
4          foreach A ∈ SolverList:
5              if (resolve_safety(A, id, id))
6                  commit(id)
7                  break
8
9  function resolve_safety(A, id, e_j)
10     S = callSolver(A)
11     Target = derive_coms(ensemble)
12     foreach t ∈ Target:
13         t.solution = rpc (resolve_safety(A, t, id))
14         if (t.solution == false)
15             S = ∅
16             break
17     if (S ≠ ∅)
18         store(S, Target)
19         return true
20     else
21         return false
22
23 function commit(r_id)
24     foreach t ∈ r_id.Target
25         commit(t)
26     update(S)
```

Fig. 11. Code snippets from Planning and Execution state.

chosen for safety resolution, *id* is the identifier of the entity that executes the algorithm to generate a solution and *e_j* is the identifier of the entity that triggered the algorithm in the entity that executes it. Initially $resolve\_safety(A, id, e\_j)$ is called locally by the ensemble leader so it has the form $resolve\_safety(A, id, id)$ (line 5).

The function *resolve_safety* includes the following. First, the appropriate solver is called to generate a solution S based on the algorithm A (line 10). Then, a corresponding communication is derived using *derive_coms* for each of the reachable entities in the ensemble. In other words, *derive_coms* finds all the children in the problem resolution tree reachable from the corresponding entity in the final ensemble.

For each identified entity, the function *resolve_safety* is called with the algorithm A (line 13) and a solution S is calculated.

The function *resolve_safety* returns a boolean:

- it returns true when the entity and all its children in the problem resolution tree contain the algorithm A in its knowledge base. If that is the case, the generated solution S is stored locally (function store (line 18));
- it returns false when the entity itself or one of its children (targets) does not contain the algorithm A in its knowledge base.

In the end the *resolve_safety* for the ensemble leader (line 5) returns true if all the entities in the ensemble are able and agreed to perform some algorithm A. If that is not the case, the ensemble leader goes through the other algorithms in the *SolverList* and repeats the process until finds a suitable algorithm A for which all ensemble participants are able to perform it. If it finds one, it calls locally the function *commit* and breaks the loop. Execution of the commit function is when the entity enters the *Execution* state in Fig. 10.

The *commit* function lines 23–26 is recursive, it goes through each entity in the ensemble and enacts a distributed commit of the best solution. It updates the solution S in each entity by updating its behaviour tree with the function update(S) (line 26). When the entity finishes with the *commit* function in the *Execution* state, it goes to *Discover*.

6.2. Correctness and completeness

In order to prove correctness of the safety resolution approach we need to prove the correctness of the safety resolution algorithm. In order to prove the correctness of the safety resolution resolution algorithm we need to analyze: (i) the protocol for on-the-fly ensemble formation and (ii) the recursive function for selecting and committing a solution. We define correctness as follows.

Theorem 3 (Correctness). *If an entity e triggers a safety problem P_i, then the Safety Problem Resolution Algorithm computes a solution S_i where the safety invariants in Definition 11 are always satisfied.*

Proof. Our assumption is that each of the entities in the safety collective adaptation process contains at least one safety solver (collision avoidance protocol) that is compatible to the solvers of the other entities in the ensemble. This means that we assume that when an ensemble is formed, each of the entities participating in the ensemble contains an appropriate solver that can generate a solution that will satisfy the safety invariant defined in Definition 11. Therefore, to guarantee correctness of the safety resolution process we just need to prove correctness of the protocol for on-the-fly ensemble formation.

In order to prove the correctness of the protocol for on-the-fly ensemble formation, we verify (using model-checking) the following properties about the coordination protocol: (1) at each point of time there won't be any ensembles (up to 16 entities) that are in *planning&execution* phase and that have overlapping $conflict\_region$ CR; (2) the highest safety index computed by the safety resolution algorithm is consistent across all entities in an ensemble that is in *planning&execution* phase. We formally specify them as follows. We denote with T the total mission execution time and with E^{τ}_{pe} the set of all ensembles in the system that are in *planning&execution* phase at time τ. Hence, we define (1) as: $\forall \tau \in T \&\& \forall es_i, es_j \in E^{\tau}_{pe}; es_i.CR \cap es_j.CR = \emptyset$ and we define (2) for an $es \in E^{\tau}_{pe}$ as: $\forall e_i, e_j \in es; e_i.s\_id = e_j.s\_ = id$.

However, the safety resolution algorithm by itself is not complete due to the following reason: each entity in a stable ensemble might not have a solver compatible to the solver proposed by the leader. However, for completeness of the approach, as we mentioned earlier, we assume that each entity in the system has at least one safety solver that is compatible to the solvers of the rest of the system. We consider this assumption reasonable because we consider safety independently from the mission and in this way, we can reason about safety before the start of the mission. That is how we guarantee safety during the whole mission execution. □

7. Related work

MMRSs are just one application domain from the variety of Cyber-physical systems (CPSs). In the literature, a lot of work has been done to address run-time adaptation of CPSs in different application domains and different levels of abstraction. Specifically, the authors of [18] present an approach to support the adaptation process of CPS based on run-time generation of verified system configurations, while in [19] the temporal costs of an autonomic manager that performs on-line verification for a specific application are analysed. Moreover, [20] presents an architecture of a middleware that supports time-deterministic reconfiguration in distributed soft real-time environments, while [21] evaluates reinforcement learning adaptation policies through a set of experiments. All the aforementioned works present general approaches that have been applied in a specific application scenario. In comparison, in this work we focus on the specificities of the domain of mobile multi-robot systems: (i) agents have partial knowledge of the system and the environment and (ii) the interplay between (possibly) partial resolution of mission problems vs. full resolution of safety problems is crucial.

In the following, we focus on mobile multi-robot systems and we review recent work on run-time approaches that support mission execution. We investigate what kind of run-time collective approaches are proposed for the different multi-robot systems and in which type of environments they are applicable. Furthermore, we focus at identifying and classifying approaches that address safety. Our aim is to understand how safety is managed and what are the constraints and limitations of

Table 1
Literature comparison.

Work	System & environment characteristics						Safety Management		Adaptation mechanisms		
	A. Type	Openness	Ens. type	Mgmt.	Hierarchy	ENV	Coop. Mechanisms	Concerns sep.	Adapt	Type	Human
[22]	Heterogenous	N/A	dynamic	N/A	N/A	N/A	cooperative	No	Yes	Both	No
[23]	Heterogenous	Yes	N/A	N/A	No	dynamic	local	No	Yes	isolated	No
[24]	Heterogenous	No	N/A	N/A	N/A	dynamic	cooperative	No	No	N/A	N/A
[25]	Heterogenous	No	dynamic	distributed	No	static	cooperative	No	Yes	isolated	No
[26]	Homogeneous	No	N/A	N/A	No	static	local	No	Yes	isolated	No
[27]	Homogeneous	No	N/A	N/A	No	static	cooperative	No	No	N/A	NO
[2]	Heterogenous	No	N/A	N/A	NO	dynamic	centralized	No	Yes	N/A	N/A
[1]	Homogeneous	No	dynamic	centralized	No	N/A	centralized	No	No	N/A	N/A
[28]	Heterogeneous	Yes	dynamic	distributed	Yes	dynamic	N/A	N/A	Yes	BOTH	No

existing methodologies addressing safety. Additionally, we consider if the approach supports adaptation on the system and which adaptation decisions could be made at run-time.

In Table 1, we note some of these run-time mission execution approaches for mobile multi-robot systems and categorize them according to the following criteria:

1) System & environment characteristics;
2) Safety management;
3) Adaptation mechanisms.

For each criteria we identify several parameters. For each of them, we categorize the approaches as follows.

System & environment characteristics

- *Agent type*: whether the considered agents in the system are homogeneous or heterogeneous.
- *Openness*: whether the approach supports systems that can accept external agents at run-time (e.g., new robots entering the mission).
- *Ensemble type*: whether the proposed approach supports formation of an ensemble structure (group of agents) that can change at run-time (dynamic) or not (static).
- *Ensemble management*: whether an ensemble is managed in a centralised or in a distributed way.
- *Hierarchy*: whether the approach provides a mechanism for hierarchical structure of ensembles (e.g. an ensemble of ensembles).
- *Environment* (dynamic vs. static): whether the approach supports modeling of systems that operate in environment that can change at run-time (e.g., moving obstacles, some other elements outside of the system can change their status).

Safety Management

- *Cooperation mechanisms*: whether the approach allows safety mechanisms that involve cooperation between different agents rather than centralized management of safety entity or local management on single robots, without any cooperation.
  It is *LOCAL* if safety mechanisms are conceived to work on single robots, without any cooperation, *CENTRALIZED* if the knowledge of the overall system is maintained by a centralized entity, or *COOPERATIVE* if there are mechanisms to share knowledge between different robots that take part in the mission.
- *Separation of concerns*: whether the approach keeps the management of safety-specific issues (e.g. safety rules) separated from the management of mission-specific issues.

Adaptation decisions

- *Adaptability*: whether the approach supports MMRSs that can adapt.
- *Type*: whether the approach supports a collective adaptation where a collection of autonomous agents collaborate together to satisfy a particular goal or isolated adaptation where one agent adapts independently from the rest of the system.
- *Human Controllability*: whether the approach enables an operator (human) to be involved in the adaptation process.

As shown in Table 1, most of the approaches are unable to deal with open systems (only 2/9 approaches are able to deal with open systems). By open systems, we mean systems that can accept external entities at run-time (e.g., new robots or new human actors). This implies that most of the approaches that have been proposed do not consider that the system evolves in terms of addition or removal of robots and/or other types of agents, including humans. This is indeed an interesting research direction since systems of the near future will be necessarily characterized by openness, and it is often impossible to assess at design time the exact boundaries and topology of the system.

A peculiar system characteristic is the capability of managing teams consisting of robots of different types (e.g., robots for grabbing objects, for video streaming, sensing and discovering relevant information). According to Table 1 most of the analyzed systems have the capability of managing heterogeneous robots, which is the direction in which we are going with our approach.

In order to manage different unpredictable situations of missions and considering situations where there is only partial communication between different agents, it is preferable that the MMR system is capable of grouping and regrouping agents in ensembles at run-time in a decentralized fashion. Most of the approaches propose solutions where they assume that all robots in the system will be able to communicate to each other. Only 2/9 approaches discuss about the possible benefits of distributed ensemble formation. The concept of ensemble enables single agents to take part in a group where they will follow certain rules and in return the ensemble offers certain advantages with respect to a preservation of a particular system quality. In this context, we don't need to consider the whole system to analyze if a particular system quality is satisfied, we only consider part of it.

Furthermore, as shown in Table 1, in all approaches the management of safety-specific issues (e.g., safety rules) is not kept separated from the functional management of the robots (e.g., the mission). Keeping a separation of concerns means for instance that the approach prescribes a special layer for managing safety, which is totally separated from the rest of the system.

Regarding the cooperation mechanisms, most of the approaches adopt local safety mechanisms, i.e. safety mechanisms that are conceived to work on single robots, without any cooperation. Centralized safety management mechanism means that there exists an entity managing the safety aspect of the overall system. As can be seen in Table 1 only 2 approaches have a centralized safety management mechanism. Instead, 4 approaches rely on co-operative safety mechanisms, meaning that safety mechanisms involve a cooperation between different robots.

Regarding the adaptation mechanisms, most of the approaches allow the system to adapt at run-time, meaning that the system is able to adapt (e.g., behaviour adaptation, trajectory recalculation, goal renegotiation, etc.) in order to find a solution depending on some change in

the context. Adaptability might be considered in conjunction with context awareness since awareness of the context is a required capability in order to support adaptability. Human involvement in the adaptation process brings some degree of control-ability that can help in defining regulations and rules about the responsibilities of the operators in operational scenarios and make adaptation more practical and safer. None of the approaches in Table 1 includes the human as a factor in the adaptation process. Regarding the collectiveness of the adaptation process, 3 of the approaches consider isolated adaptation where the agent adapts its behaviour independently from the rest of the system, while only 2 approaches consider the two types of adaptation: (i) on a collective level where multiple agents must adapt altogether and transactionally and (ii) isolated adaptation where one agent adapts its behaviour independently from the rest of the system.

Classifying our approach in this classification schema is as follows: It supports modeling of *open systems* which consist of *heterogeneous agents* that are *context-aware* about their operational context. The agents can be grouped in *dynamic ensembles*, which are groups of agents that are managed in a *distributed way* among the ensemble participants. Furthermore, the approach allows for a *hierarchical structure of ensembles* for satisfying a particular goal as we showed in [12]. Regarding the environment, the approach has mechanisms for modeling MMRSs that operate in *dynamic environments* and have some degree of *unpredictability* (ex. birds flying, animals walking, etc).

Most of the approaches we have analyzed do not consider safety aspects separate from the functional behaviour of the robots. In our approach we make a *clear separation of concerns between safety and mission* aspects. We consider this as extremely important for managing complex missions and it is one of the fundamental parts on which we base our work. This way an operator modeling its mission can focus on the mission specification, while a safety engineer can focus on the safety-specific mechanisms, thus making safety-specific mechanisms reusable across missions, projects, and organizations.

Another really important feature of our approach is allowing *cooperation mechanisms* between different agents when safety-related issues are triggered. This means that safety is not managed in a centralized way (there isn't an entity that manages the whole aspect of safety, but it is managed on a level of ensemble in a distributed way where each agent should perform an appropriate behavior as part of the ensemble).

Moreover, our framework contains structures that enable system designers to design systems which are *adaptable* during mission execution. It allows part of the adaptation decisions to be done at *run-time*. That being said, we make clear distinction about which decisions should be made at design-time versus decisions at run-time. Regarding the type, we allow agents to adapt on a collective (ensemble) level, which means that a collection of autonomous agents collaborate to perform adaptation in order to satisfy a particular goal or solve a particular problem. In the end, we allowed the operator (human) be able to have control in the adaptation process. Some adaptation decisions can not be done by the system at run-time, however with our approach as we showed in [12] we allow the operator to take over and participate as agent in the adaptation process.

8. Conclusions

In this paper, we presented a collective adaptation approach that consists of two parts: one for (potentially partial) resolution of mission problems and one for full safety resolution i.e. one that ensures a full satisfaction of safety invariants. While most of the proposed solutions for collective adaptation work under the assumption that all the knowledge used to adapt a system is fully specified at design time (i.e., a predefined set of issues) and is centrally controlled by a specific component (i.e., a set of predefined solvers), our approach, as depicted previously, addresses collective adaptation problems in a decentralized fashion, at run-time, with new solvers that can be introduced at any time. At the same time, in highly dynamic and distributed environments, our approach provides a way to dynamically understand which parts of the system should be selected to help solve an adaptation issue making a clear distinction when resolving mission vs. safety-related issues. That way, we can ensure full satisfaction of safety, while guaranteeing (potentially partial) mission completion.

Currently, we are performing an extensive experimental campaign to evaluate the collective adaptation process (CAP) by simulation. Simulation is performed by using a Software-In-The-Loop (SITL) platform. We are measuring mission satisfiability that gives information on how much percentage of the mission is performed by the system taking in consideration various application domains. We are using heterogeneous robots operating under various circumstances (e.g. different number of tasks to be performed, different size of MMRSs, different number of problems triggered during mission execution, different ratio between safety and mission problems triggered during mission execution etc.). The final goal is to find out if our collective adaptation approach is scalable for managing real-sized missions. Additional materials concerning the experimental setup and the results is available here: https://darkobozhinoski.github.io/MMRS/.

Moreover, we plan to integrate the approach with a suitable extension of the FLYAQ platform [3]. This platform permits to graphically define civilian missions for a team of autonomous multicopters via a domain specific language to make the specification of missions accessible to people with no expertise in IT and robotics.

Acknowledgments

Research partly supported from the EU H2020 Research and Innovation Programme under GA No. 731869 (Co4Robots) and from the European Research Council (ERC) under GA No. 681872 (DEMIURGE).

References

[1] A. Desai, E.A. Cappo, N. Michael, Dynamically feasible and safe shape transitions for teams of aerial robots, in: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2016, pp. 5489–5494.
[2] J.A. DeCastro, J. Alonso-Mora, V. Raman, D. Rus, H. Kress-Gazit, Collision-free reactive mission and motion planning for multi-robot systems, in: Proceedings of the Robotics Research, Springer, 2018, pp. 459–476.
[3] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, M. Tivoli, Flyaq: Enabling non-expert users to specify and generate missions of autonomous multicopters, in: Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 801–806.
[4] D. Di Ruscio, I. Malavolta, P. Pelliccione, Engineering a platform for mission planning of autonomous and resilient quadrotors, in: Proceedings of the International Workshop on Software Engineering for Resilient Systems, Springer, 2013, pp. 33–47.
[5] T. Skrzypietz, Unmanned aircraft systems for civilian missions, BIGS policy paper: Brandenburgisches Institut für Gesellschaft und Sicherheit, BIGS, 2012.
[6] D.D. Ruscio, I. Malavolta, P. Pelliccione, M. Tivoli, Automatic generation of detailed flight plans from high-level mission descriptions, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, ACM, 2016, pp. 45–55.
[7] M. Lahijanian, M.R. Maly, D. Fried, L.E. Kavraki, H. Kress-Gazit, M.Y. Vardi, Iterative temporal planning in uncertain environments with partial satisfaction guarantees, IEEE Trans. Rob. 32 (3) (2016) 583–599.
[8] R.G. Dromey, From requirements to design: Formalizing the key steps, in: Proceedings of the First International Conference on Software Engineering and Formal Methods, IEEE, 2003, pp. 2–11.
[9] A. Bucchiarone, C.A. Mezzina, M. Pistore, H. Raik, G. Valetto, Collective adaptation in process-based systems, in: Proceedings of the IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems (SASO), IEEE, 2014, pp. 151–156.
[10] M. Colledanchise, P. Ögren, How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees, IEEE Trans. Rob. 33 (2) (2017) 372–389.
[11] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Comput. (Long Beach Calif) 36 (1) (2003) 41–50.
[12] D. Bozhinoski, A. Bucchiarone, I. Malavolta, A. Marconi, P. Pelliccione, Leveraging collective run-time adaptation for uav-based systems, in: Proceedings of the 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2016, pp. 214–221.
[13] T. Bures, F. Krijt, F. Plasil, P. Hnetynka, Z. Jiracek, Towards intelligent ensembles, in: Proceedings of the European Conference on Software Architecture Workshops, ACM, 2015, p. 17.
[14] P language Manual. [Online]. Available: https://github.com/p-org/P/blob/master/Doc/Manual/pmanual.pdf.

[15] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, Y. Xie, Zing: A model checker for concurrent software, in: Proceedings of the Computer Aided Verification, Springer, 2004, pp. 28–32.
[16] P.R. Lewis, M. Platzner, B. Rinner, J. Tørresen, X. Yao, Self-Aware Computing Systems, Springer, 2016.
[17] M. Wirsing, M. Hölzl, N. Koch, P. Mayer, Software Engineering for Collective Autonomic Systems: The ASCENS Approach, 8998, Springer, 2015.
[18] M. García-Valls, D. Perez-Palacin, R. Mirandola, Pragmatic cyber-physical systems design based on parametric models, J. Syst. Softw. (2018).
[19] M.M. Bersani, M. García-Valls, Online verification in cyber-physical systems: practical bounds for meaningful temporal costs, J. Softw. Evolut. Process 30 (3) (2018).
[20] G. Valls, I.R. López, L.F. Villar, Iland: an enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems, IEEE Trans. Ind. Inf. 9 (1) (2013) 228–236.
[21] J. Panerati, F. Sironi, M. Carminati, M. Maggio, G. Beltrame, P.J. Gmytrasiewicz, D. Sciuto, M.D. Santambrogio, On self-adaptive resource allocation through reinforcement learning, in: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), IEEE, 2013, pp. 23–30.
[22] Y. Cui, R.M. Voyles, J.T. Lane, M.H. Mahoor, Refresh: a self-adaptation framework to support fault tolerance in field mobile robots, in: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014), IEEE, 2014, pp. 1576–1582.
[23] L.E. Parker, Alliance: an architecture for fault tolerant multirobot cooperation, IEEE Trans. Robot. Autom. 14 (2) (1998) 220–240.
[24] M.G. Morais, F.R. Meneguzzi, R.H. Bordini, A.M. Amory, Distributed fault diagnosis for multiple mobile robots using an agent programming language, in: Proceedings of the International Conference on Advanced Robotics (ICAR), IEEE, 2015, pp. 395–400.
[25] M.T. Khan, M. Qadir, F. Nasir, C. de Silva, A framework for a fault tolerant multi-robot system, in: Proceedings of the 10th International Conference on Computer Science & Education (ICCSE), IEEE, 2015, pp. 197–201.
[26] E. Castello, T. Yamamoto, F. Dalla Libera, W. Liu, A.F. Winfield, Y. Nakamura, H. Ishiguro, Adaptive foraging for simulated and real robotic swarms: the dynamical response threshold approach, Swarm Intell. 10 (1) (2016) 1–31.
[27] A. Desai, I. Saha, J. Yang, S. Qadeer, S.A. Seshia, Drona: a framework for safe distributed mobile robotics, in: Proceedings of the 8th International Conference on Cyber-Physical Systems, ACM, 2017, pp. 239–248.
[28] Y.A. Alrahman, R. De Nicola, M. Loreti, On the power of attribute-based communication, in: Proceedings of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems, Springer, 2016, pp. 1–18.

Darko Bozhinoski is a Postdoctoral researcher at IRIDIA, Universite Libre de Bruxelles (Brussels, Belgium). He is a part of the DEMIURGE project: project funded by the European Research Concill that focuses on automatic design of robot swarms. His research topics are mainly in software engineering, formal methods, swarm robotics and self-adaptive systems. He received a Ph.D. in Computer Science from Gran Sasso Science Institute in 2017. In his Ph.D. thesis, he is proposing a formal modeling framework that enables Mobile Multi-Robot systems (MMRSs) to perform collective adaptation in a decentralized-fashion at run-time, guaranteeing preservation of safety constraints. In 2016 he was awarded the Fulbright Scholarship that enabled him to spend the academic year 2016–2017 at Carnegie Mellon University as a visiting research scholar. More information is available at: http://iridia.ulb.ac.be/~dbozhin/.

David Garlan is a Professor of Computer Science in the School of Computer Science at Carnegie Mellon University. He received his Ph.D. from Carnegie Mellon in 1987 and worked as a software architect in industry between 1987 and 1990. His interests include software architecture, self-adaptive systems, formal methods, and cyber-physical systems. He is considered to be one of the founders of the field of software architecture, and, in particular, formal representation and analysis of architectural designs. He is a co-author of two books on software architecture: “Software Architecture: Perspectives on an Emerging Discipline”, and “Documenting Software Architecture: Views and Beyond.” In 2005 he received a Stevens Award Citation for “fundamental contributions to the development and understanding of software architecture as a discipline in software engineering.” In 2011 he received the Outstanding Research award from ACM SIGSOFT for “significant and lasting software engineering research contributions through the development and promotion of software architecture.” He is a Fellow of the IEEE and ACM. More information is available at: http://www.cs.cmu.edu/~garlan/.

Ivano Malavolta is Assistant Professor at the Vrije Universiteit Amsterdam, The Netherlands, Department of Computer Science, Faculty of Sciences. His research focuses on data-driven software engineering, software architecture, model-driven engineering (MDE), and mobile-enabled systems. Recently, he is applying empirical methods to assess practices and trends in the field of software engineering. He authored more than 70 papers in international journals and peer-reviewed international conferences proceedings; they include articles published in the IEEE Transactions on Software Engineering (TSE) and the Interna-tional Conference on Software Engineering (ICSE), which are considered the leading journal and conference in the field of software engineering, respectively. He received a PhD in computer science from the University of L’Aquila in 2012. He is a member of ACM and IEEE. More information is available at http://www.ivanomalavolta.com.

Patrizio Pelliccione is Associate Professor at the Chalmers University of Technology and University of Gothenburg, Sweden, Department of Computer Science and Engineering. He got his Ph.D. in 2005 at the University of L’Aquila (Italy) and from February 1, 2014 he is Docent in Software Engineering, title given by the University of Gothenburg. His research topics are mainly in software engineering, software architectures modelling and verification, autonomous systems, and formal methods. He has co-authored more than 100 publications in journals and international conferences and workshops in these topics. He has been on the program committees for several top conferences, and is a reviewer for top journals in the software engineering domain. He is very active in European and National projects. In his research activity he has collaborated with several industries such as Volvo Cars, Volvo AB, Ericsson, Jeppesen, Axis communication, Thales Italia, Selex Marconi telecommunications, Siemens, Saab, TERMA, etc. More information is available at http://www.patriziopelliccione.com.