2016 42th Euromicro Conference on Software Engineering and Advanced Applications

Leveraging Collective Run-time Adaptation for UAV-based Systems

Darko Bozhinoski*, Antonio Bucchiarone†, Ivano Malavolta‡, Annapaola Marconi† and Patrizio Pelliccione§
*Gran Sasso Science Institute, L'Aquila, Italy darko.bozhinoski@gssi.infn.it
†Fondazione Bruno Kessler, Trento, Italy, [bucchiarone,marconi]@fbk.eu
‡Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, i.malavolta@vu.nl
§Chalmers University of Technology | University of Gothenburg, Gothenburg, Sweden, patrizio.pelliccione@gu.se

***Abstract*—UAV-based systems are systems that are composed of a team of drones, various devices (like movable cameras, sensors), and human agents, which collaborate each other to accomplish defined missions. Since humans are constituent part of these systems, UAV-based systems are both mission-critical and safety-critical. Moreover, these systems are requested to operate in potentially unpredictable and unknown environments. A model of the environment describing, e.g. obstacles, no-fly zones, wind and weather conditions might be available, however, the assumption that such model is both correct and complete is often wrong.**

In this paper, we describe a novel approach for managing the run-time adaptation of UAV-based systems. Our approach is based on a generic collective adaptation engine that addresses collective adaptation problems in a decentralized fashion, operates at run-time, and enables the addition of new entities at any time. Moreover, our approach dynamically understands which parts of the system should be selected to solve an adaptation issue. The feasibility and scalability of the approach have been empirically evaluated in the context of a private company surveillance scenario.

I. Introduction

Unmanned Autonomous Systems (UAVs) are increasingly attracting attention as instruments to automate and reduce the human involvement in scenarios involving repetitive or dangerous tasks (ex. natural catastrophes, delivery services, surveillance, environmental monitoring). UAV-based systems are composed of a team of drones, various devices (like movable cameras and sensors), and human agents.

Designing software for UAVs in order to operate in unknown disaster scenarios is difficult. Possible scenarios where UAVs operate may vary in many different aspects, like the environment, the scale of the affected area, the type of mission they need to perform. Environments in which UAV-based systems operate are often unpredictable and unknown; because of that UAVs should be able to deal at run-time with unknown situations that cannot be anticipated completely at design-time. A model of the environment (containing for example obstacles, no-fly zones, wind and weather conditions) might be available, however, we cannot assume that such model will be always both correct and complete.

This paper describes a novel approach for managing the run-time adaptation of UAV-based systems. The approach is based on a generic collective adaptation engine that, operating at run-time, addresses collective adaptation problems in a decentralized fashion. New entities (in literature sometime referred as agents) can be introduced at any time during the mission execution in order to ensure the satisfaction of the mission. Once an adaptation issue is triggered, our approach provides a way to dynamically understand which parts of the system should be selected in order to solve such issue. Our strategy is to organize the UAV-based system in different levels and to create a mechanism that decides the right scope for every adaptation issue. Our model of collective adaptation is built around the concept of *ensemble*, i.e. a collection of autonomous entities that collaborate to perform certain tasks.

The overall execution time and memory consumption of the approach have been empirically evaluated in the context of a surveillance scenario of the premises of a company. The scenario involves various agents like people (guards, maintainers of the equipment), movable cameras, fixed sensors, UAVs, ground stations and a central station. The goal of the performed experiment is to assess (i) if our collective adaptation engine can be used at run-time to manage the adaptation of UAV-based missions and (ii) its scalability for managing real-sized missions.

The rest of the paper is organized as follows: Section II describes our motivating scenario, Section III presents our approach for collective run-time adaptation for UAV-based systems, and Section IV shows our empirical evaluation of the approach. Finally, related work is reviewed in Section V, and Section VI closes the paper.

II. Motivating Scenario: a surveillance system

The motivating scenario and running example of this paper is a surveillance system for the premises of a private company. The entities involved in our scenario are the following:
Guards - persons that observe the surveillance process located in one of the buildings. When an intruder enters, a guard can physically approach him.
Maintainers - persons that are in charge of maintaining the equipment (i.e. drones, cameras, or sensors). They can make changes in the equipment: replacing old with new, adding new ones (e.g. add new drones or cameras), etc.
UAVs - entities that follow specific protocols defined in the service agreement of the company. They can accomplish several tasks.
Movable Cameras - entities placed on the top of the building moving along its edges.

2376-9505/16 $31.00 © 2016 IEEE
DOI 10.1109/SEAA.2016.41

Fixed Sensors - entities placed in strategic places to monitor movements in the environment.
Ground Stations - entities that receive telemetry data and can do simple recalculations of the mission. A single drone can be managed by a single ground station, while one ground station can manage multiple drones.
Central Station - entity located somewhere in the cloud, where surveillance videos are stored. Thanks to the central station guards can access video streams from a particular camera, sensor or drone in order to check a particular situation.

In normal operation mode, we have several drones patrolling the area, cameras moving along the edges of the buildings, fixed sensors located in strategic places sensing for a suspicious behaviour and a guard observing the whole process located in one of the buildings. If a suspicious behaviour is noted, the entities should perform as follows. First, one of the drones takes a picture of the intruder. Then, a drone delivers a warning voice message to the intruder regarding the actions that will be taken towards him. In parallel with this, notification is send to the guard about a suspicious behaviour.

Considering the scenario described above, the drones should be able to perform the following actions: (i) patrol the area searching for suspicious behaviour; (ii) once an intruder is found it will illuminate the area for a picture to be taken; (iii) take picture of the intruder; and finally (iv) communicate with the intruder by giving a warning message.

The number of entities in the system can vary according to the size of the area, number of buildings and other caracteristics of the specific company that wants to use the surveillance system.

III. Collective run-time Adaptation for UAV-based systems

UAV-based systems are both safety-critical and mission-critical since UAVs collaborate with humans and other devices to accomplish defined missions. Missions are defined at design-time and include the information that is available at that time. However, a proper management of the run-time phase is required since environments in which these systems have to operate are often unpredictable and unknown.

In this paper we focus on the run-time phase. A possible design-time phase for the specific example can build on top of the FLYAQ [1], [2] 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 this paper we assume that each entity implements the MAPE loop and more precisely implements the state machine in Figure 1. Moreover, the implementation of each entity should come with information about the solvers it provides and the issues it triggers. It is important to note that also humans (indirectly) follow the MAPE loop because their interaction with the rest of the system, often realized via devices (e.g., a smartphone), is required to follow an interaction protocol compliant with the MAPE loop.

The solution involves various entities including UAVs, sensors, cameras, and also humans (via their devices) that collaborate as an ensemble. In the scenario described in Section II we make use of the following entities: UAVs (i.e., drones in our case), movable cameras, fixed sensors, humans (via their devices), precisely guards and maintainers, ground stations and a central station. Each entity can be part of different ensembles and can produce issues and provide solutions.

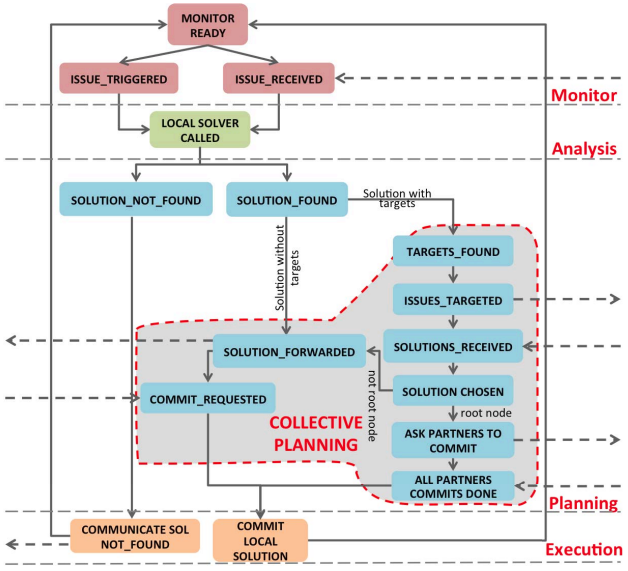

Fig. 1. State machine animating each entity of the MAPE loop hierarchy

Entities might be organized in several possible topologies according to the needs of the particular domain. Figure 2 shows the topology of the example. It is hierarchical with the central station, i.e. CS, coordinating the other entities. CS together with his sub-entities, i.e. three ground stations, named $H1$, $H2$, and $H3$, and a guard (human) called $G1$, forms the $E4$ ensemble. $H1$ manages the ensemble $E1$ that includes also drones $D1$ and $D2$, camera $C1$ and the maintainer $M1$. $H2$ forms the ensemble $E2$ together with drones $D3$ and $D4$ and maintainer $M2$. $H3$ manages the ensemble $E3$ that includes also drone $D5$, camera $C3$ and the maintainer $M3$.

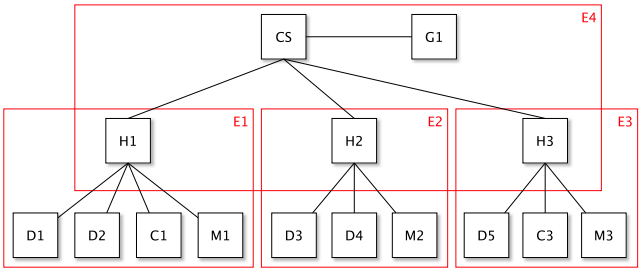

Fig. 2. Types of entities and communication topology

Figure 1 shows the state machine that each entity should implement.

During its normal execution, each entity is in the Monitoring phase (`Monitor Ready` state), while monitoring the environment. By now, issues can come both by the entity itself (`Issue Triggered` state), and by a different entity, which asks for solving an issue (`Issue Received` state). At this point, the corresponding solver is run (`Local Solver Called` state). Here we enter in the Planning phase, where two directions can be taken, depending on the solver's response. If the solver is not able to find a solution (`Solution Not Found` state), the Execution phase is triggered (`Communicate Solution Not Found` state) and the failure is forwarded to the monitor of the entity that generated the issue. Otherwise, the solver has found a solution (`Solution Found` state). In this case, the *collective planning* phase starts; all the entities involved in the issue resolution process will collectively collaborate to solve the issue. Here, two different situations can be observed. The solution provided by the solver does not involve other entities. The entity simply sends information about its capability to solve the current issue (`Solution Forwarded` state) and it waits for a commit request (`Commit Requested` state). Differently, the solution provided by the solver foresees the involvement of other entities, called targets, which are firstly found (`Targets Found` state), and then triggered (`Issues Targeted` state) to be involved in the resolution process. Once the current entity receives feedback from the triggered entities (`Solution Received` state), it selects the most appropriate solution (`Solution Chosen` state). From here on, we should distinguish two cases. If the issue was triggered internally (root node), the entity first commits all dependent solutions (`Ask Partners to Commit` state), wait that they are all committed (refexAll Partners Commits Done state), and then it commits its local solution (`Commit Local Solution` state). Otherwise, if the issue was coming from outside, the entity reports the feedback to the issue sender (`Solution Forwarded` state), and then it waits for a future commit (`Commit Requested` state). The entity can receive a positive or a negative reply for its proposed solution. In both cases, it executes a solution commit (`Commit Local Solution` state), which results to be empty in the negative case.

A. Collective Adaptation

In order to explain our model of collective adaptation we start introducing the notion of *entity*. An entity can be seen as a representation of a computational or human actor that can be part of multiple ensembles.

The model of an entity in an ensemble is primarily determined by the ways it collaborates with other entities. Collaboration consists in managing issues and responding to issues raised by others. As such, an entity model includes a set of issues it can produce, and a set of solvers it provides.

Issues generally correspond to different critical situations that can happen to an entity of an ensemble. Each issue includes a set of parameters describing it. An issue corresponds to a particular situation occurring in an ensemble. For example in our scenario a drone can trigger an issue type $obstacleDetected = \{obstacleDistance\}$.

Collaborating in an ensemble, each entity can provide one or more solvers. In our scenario each ground station is able to recalculate the path for some drones when for example some of them has identified an obstacle. To manage this situation it has a solver $RecalculatePath = \{\langle obstacleDetected, \{BatteryLevel\}\rangle\}$ where $obstacleDetected$ represents the issue it is able to solve, while $BatteryLevel$ is a parameter that expresses the battery level of the drone involved.

An *entity* is bound to a certain entity type within an ensemble and its state is determined by its data (i.e. parameters and preferences) and by any ongoing issue resolution activities.

In the scenario, we can distinguish different entities: *D1*, *C1*, *G1*, *CS*, etc. To represent collaboration of multiple entities we introduce the notion of *ensemble*: it is a collection of autonomous entities that collaborate to perform certain tasks. In our scenario E1, E2, E3, and E4, as described in Figure 2, are examples of ensemble where $E1 = \{H1, D1, D2, C1, M1\}$, $E2 = \{H2, D3, D4, M2\}$, etc.

In our framework we have two types of activities that an entity can execute during a collective adaptation problem resolution: *issue communication* and *issue resolution*. Issue communication is used to send an issue to a *target entity* (see definition below) that is supposed to resolve it. The issue may be sent to multiple partners at a time in attempt to find a better solution. Issue communication comprises a few steps: 1) the issue is sent to all target entities; 2) the replies are received from the partners able to resolve the issue; 3) the preferable solution is chosen; 4) the preferable solution is committed. While the issue communication is a way to propagate resolution activities between partners, *issue resolution* corresponds to the high-level model of internal elaboration being done by entities. In particular, we assume that the issue may either arise internally (when the issue originally occurs in this entity) or is received by one of the entity solvers. As soon as the issue is raised, the entity may either resolve it locally or propagate issues to the other entities as a part of the resolution procedure.

The issue resolution procedure within an ensemble can be represented as a tree, which we call *issue resolution tree*. Indeed, the resolution procedure always starts from creating an issue resolution. It may instantiate further issue communications to resolve sub-issues. In turn, each issue communication may target a few entities, each of which consequently initiates an issue resolution and so on.

An issue resolution tree is a very intuitive abstraction for understanding and analysing how our approach works.

B. Issue Resolution Algorithm

To realize our approach, in Figure 3 we abstractly define an algorithm that covers the procedures for issue resolution and commitment (functions `resolve` and `commit` respectively). The function `resolve` is used recursively to trigger a distributed resolution procedure across multiple entities within an ensemble.

```
1  function resolve(i, e)
2      sol := callSolver(i)
3      foreach s.issues ∈ sol:
4          Com := derive_coms(s.issues)
5          foreach c ∈ Com
6              Target := find_targets(c)
7              S = ∅
8              s_best = null
9              foreach t ∈ Target
10                 t.solution = rpc(resolve(c.issue, t))
11                 S = S ∪ t.solution
12             s_best := AHP(S)
13     if e != root
14         store(s_best)
15     else
16         commit(s_best, Target)
17
18 function commit(s_best, Target)
19     foreach t ∈ Target
20         execute(t.best)
21     execute(s_best)
```

Fig. 3. Issues Resolution Algorithm

The function is called locally by an entity e that originally detected an issue i. Further recursive calls are propagated using a *Remote Procedure Call* (line 10). The function includes the following important steps:

Line 2. The solver of the entity e is invoked and a solution sol is calculated for the specific issue i. Function `callSolver` is beyond the scope of this paper but may generally exploit various entity-specific and domain-specific solvers.

Lines 4-11. For the solution returned, a set of sub-issues is identified. This is done with the function `derive_coms` that derives all sub-issues that must be resolved for a given solution in form of corresponding issue communications. For each issue communication, the set of potential solvers is identified across all reachable entities (function `find_targets`). Finally, to understand how well the targets can handle sub-issues, the `resolve` function is called remotely on the targets (line 10) and all the possible solutions are memorized in S.

Line 12. Once the solutions to sub-issues are obtained from the remote entities, the Analytic Hierarchy Process (AHP) algorithm [3] is executed to identify the best solution s_{best}.

Lines 13-17. If the current entity is not the resolution tree root, the best solution is stored locally (function `store`). If the current entity is the resolution tree root (line 16), `commit` is executed. Function `commit` (line 18) enacts a *distributed commit* of the best solution. It takes as input the best solution and the set of involved entities in the specific issue resolution. It includes the following steps:

Lines 19-20. $Execute$ is called for each of the target entities corresponding to the best solution. Execute is an asynchronous function, so as not to impede the solution execution on line 21.

Line 21. Entity e executes the internal solution corresponding to the best solution.

C. Application to the surveillance scenario

According to the scenario described in Section II, during the system's normal execution, the following issues may be raised.

(I1) Equipment fault detected - Entities that can raise this issue are: (i) cameras, (ii) drones, (iii) sensors, (iv) ground station.

(I2) Possible Collision between drones - Entities that can raise this issue are: drones and ground station.

(I3) Low Battery - Entities that can raise this issue are: cameras, drones, sensors, ground station.

(I4) Intrusion detected - Entities that can raise this issue are: cameras, drones, sensors. This is a general issue. For a resolution, more specific sub-issues should be defined. For each of the actions defined in the service agreement a sub-issue is raised. From our scenario defined above, we can define I4 as a set of the following 4 sub-issues:

(I4.1) The area where the intruder is detected should be lit
(I4.2) Photo of the area should be taken
(I4.3) Notification to the guard should be send
(I4.4) Warning message to the intruder should be send

(I5) Obstacles detected - Entities that can raise this issue are: drones and ground station.

(I6) Mission incomplete - This issue can be raised as a consequence/result of the previous six issues.

For each of the issues mentioned above, the entities provide specific solvers that solve a specific issue. In our scenario we have the following solvers.

(S1) Stop an entity - provided by maintainer and guard
(S2) Start new entity - provided by maintainer and guard
(S3) Manually guide the drone - provided by guard
(S4) Recalculate path - provided by ground station and central station.
(S5) Update planned trajectory - provided by drones
(S6) Recalculate mission - provided by the central station
(S7) Illuminate the area - provided by drones
(S8) Take a photo - provided by drones
(S9) Communicate with intruder (send voice message) - provided by drones and guard
(S10) Send notification to guard - provided by the central station
(S11) Replace equipment - provided by the maintainer

Here we will describe how the resolution happens. We will go through each of the issues and we will describe which solvers are appropriate for the different issues.

I1: When some of the entities (camera, sensor, drone, ground station) have a fault, the issue **Fault detected** is raised. In parallel, the issue **Mission incomplete (I6)** is raised as well. Here, the maintainer can solve the first issue (I1) by providing the solver S1 (stopping the entity).

I2: In parallel with this issue, the issue **Mission incomplete (I6)** may be raised. In order to resolve possible collision between drones we have three possible resolutions. First, the drone itself can provide solution - S5. Second, the ground station or the central station can recalculate the path of the drones that are included in the possible collision - S4. Depending on the computation that needs to be performed (central station has higher computation power) one of the solutions is chosen. Third, the guard can manually guide the drones - S3. In choosing the solution important role plays the

distance between the drones and their speed. If there is enough time, central solution can provide the solution. If that is not the case and if we need prompt solution, we make simple change in the directions of the drones(both drones turn left) or the guard manually guides the drones to a safe position. If it is something in between we can choose the solution provided by the ground station.

I3: When some of the entities (camera, sensor, drone) have a low battery, the issue **Low battery** is raised. In parallel with that, the issue **Mission incomplete (I6)** is raised as well. Here, the maintainer can solve the issue I3 by providing S1 (stopping the entity).

I4: In parallel with this issue, the issue **Mission incomplete (I6)** may be raised. In order to resolve the general issue **Intruder detected** we use the formal service agreement mentioned above. For each of the actions in the agreement a sub-issue was raised. The resolution asks each of the sub-issues defined above to be resolved. In our case, I4.1 is resolved by S7, I4.2 is resolved by S8, I4.3 is resolved by S10 and I4.4 is resolved by S9. When taking in consideration the solutions between different drones we put attention to the position of the intruder, the drone's position and the position of the safety zone. Safety zone plays important role, because solution that requires from the drones to cross the safety zone have lower priority comparing to the other solutions.

I5: In parallel with the issue **Obstacles detected**, the issue **Mission incomplete (I6)** may be raised. In order to avoid the obstacles we have the following resolutions. First, the drone itself can provide solution - S5. Second, the ground station can recalculate the path taking in consideration the size of the obstacle - S4 and third, the guard can manually guide the drone - S3. In choosing the most appropriate solution the distance between the drone and the obstacle and the drone's speed play important role.

I6: To resolve this issue, which appears as a result of the previous issues we need first to implement S11. After that we use the solver S6 and in the end S2. In that point, we replaced/added an entity, we did recalculation of the mission and we started the new entity. If the issue is still not resolved, we repeat the cycle with solvers until resolution is achieved.

In the next section we show the experimental evaluation we performed on the scenario described above.

IV. Evaluation

We implemented the first version of our collective adaptation engine (named CAE in the rest of this section) in Java and we evaluated it by executing an experiment with a focus on how it performs in terms of feasibility and scalability. In our experiment we are concerned with measuring and understanding how the CAE is able to provide solutions in terms of collective adaptation strategies. In order to automatically solve an issue at the level of each entity, we adopt service composition approach presented in [4]. According to it, an issue is transformed into a planning problem and planning techniques are used to resolve it. In this paper we do not focus the attention in the correctness of the solution provided by each solver but instead in the evaluation of our collective adaptation approach.

A. Experiment Design

The **research questions** of our experiment are:

- RQ1: Is the execution time of the CAE good enough to be used at run-time to manage the adaptation of UAV-based missions?
- RQ2: Is the CAE scalable for managing real-sized missions performed by real UAVs?

The objective of these research questions is to measure and understand how the CAE is able to support run-time collective adaptation. It is important to note that we do not aim at observing the actual execution of the identified solutions (e.g., analysing the exact adapted trajectories, how obstacles are actually avoided, how photos are taken, etc.); this choice is based on the need to clearly and sharply isolate the measurements of the CAE from all the potential sources of confusion and bias coming from the actual execution of the missions. Experimentation on the full execution of real missions is in the focus of another work we are currently working on.

The experiment is designed as a multi-test within object study [5], because it is conducted on a single *object* (i.e., the current implementation of CAE) across a set of *subjects* (i.e., a set of UAV-based missions).

In the following we provide the **independent variables** of our experiment:

iv_1: number of issues raised while executing the mission;

iv_2: the machine in which the CAE is deployed and running, it can have two values:

- M: a dedicated machine running an Ubuntu 14 LTS Linux distribution with a 32-cores (each of them with a 1.4Ghz frequency) and 64Gb RAM;
- R: a Raspberry Pi 2 Model B[1] running a Raspbian 7 Linux distribution with a 900MHz quad-core ARM Cortex-A7 CPU, and 1Gb RAM.

The main idea of running our experiment on two different machines is to assess whether the current CAE implementation can be used for run-time adaptation both on highly performing and less powerful machines. In our scenario, the central and ground stations can be represented by the highly performing machine M of the experiment, whereas the drones D1-D5 can be represented by the less powerful Raspberry Pi device R of our experiment.

The **dependent variables** of our experiment are:

dv_1: execution time of the whole adaptation part of the mission in milliseconds (RQ1);

dv_2: the amount of memory used for managing all the raised issues during the mission in Mb (RQ2).

It is important to note that the measurements for assessing the values of all independent and dependent variables were performed considering exclusively the adaptation code, not

[1] https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

the actual run-time of the solvers which would have been too application-specific.

Each **treatment** models a sequence of raised issues within a mission. Each treatment is represented by a tuple $< iv_1, iv_2 >$, where we assign specific values to each independent variable within the tuple. More specifically, in each treatment iv_1 can have one of the following values: $< 1, 250, 500, 750, 1000 >$, iv_2 can be either M or R. For what concerns the types of issues, we raise issues of type I1, I2, I3, I4, and I5; we do not consider I6 issues because they are raised as a result of one of the other I1-I5 previous issues. Within each treatment, both the order of the raised issues within the sequence and the entities raising them is randomly chosen. As an example, the values $< 100, P >$ represent the treatment in which the total number of raised issues is 100, where 30 issues can be of type I1, 20 issues can be of type I2, etc. (summing up to 100 issues in total), and the experiment has been run on the Raspberry Pi device.

B. Discussion of Results

We created 1000 treatments for each single combination of all the possible values of our iv_1 and iv_2 independent variables, resulting in a total of $1000 \cdot (5 \cdot 2) = 10000$ runs of the experiment.

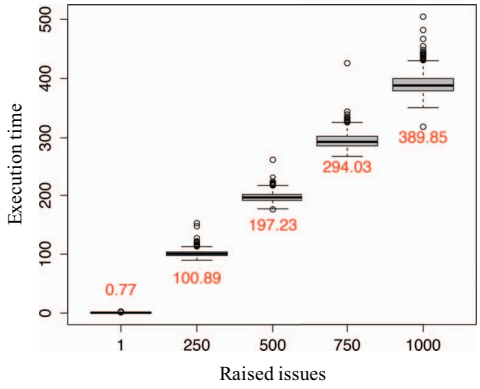

Fig. 4. Execution time on M per number of raised issues (in milliseconds)

For what concerns the **use at run-time** of the CAE to manage the adaptation of UAV-based missions (RQ1), we measured the execution times of the mission represented by each treatment. Figure 4 shows the distribution of execution times for each value we assigned to iv_1 (for each column the number in red represents the mean of all the values obtained for each class of treatments). More specifically, in Figure 4 we show how the execution time varies when the current CAE implementation is deployed and running on the highly performant machine M.

The first observation is that the execution time of whole treatment has a linear trend with respect to the number of raised issues; this is something we could expect since here we are measuring the total amount of the time that the whole treatment takes to complete. More interestingly, we can notice

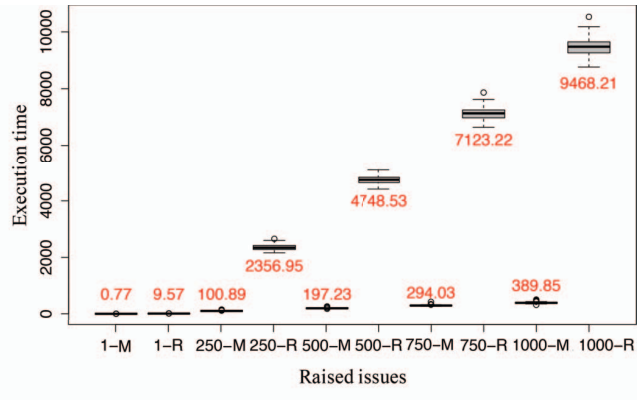

Fig. 5. Execution time per number of raised issues (in milliseconds)

that solving a single issue on M takes in average 0.77 milliseconds, whereas it takes an average of 9.57 milliseconds on the Raspberry Pi device (see Figure 5). Also, we can observe that solving a full sequence of 1000 issues on M takes an average of 389.85 milliseconds; this result is encouraging since it shows that the current implementation of CAE can solve a burst of 1000 issues in under 400 milliseconds[2]. As expected, when running on the Raspberry Pi, our CAE implementation performs worse, but still it gives highly encouraging results with an average of 9.57 milliseconds for solving a single issue (together with its sub-issues according to the resolution tree) and a maximum resolution time of 9468.21 milliseconds for solving a burst of 1000 issues. By looking at the size of the boxes in the figure we can see another interesting result, indeed the execution times of our CAE implementation has a very low variance with respect to the number and type of raised issues, meaning that our CAE is able to produce adaptation plans within a predictable (low) amount of time, thus making us reasonably confident about the possibility of using it at run-time to manage the adaptation of UAV-based missions.

For what concerns the **scalability** of the CAE for managing real-sized missions performed by real UAVs (RQ2), we decided to measure the memory consumption of the whole issue resolution algorithm. Figure 6 shows the memory used in the JVM heap for storing all the Java objects needed during the whole mission.

Obtained results tell us that for solving a single issue our CAE uses an average of 0.97 Mb of memory, whereas for solving a burst of 1000 issues to be solved all together (which we recall is an extremely pessimistic situation) it uses an average of 3.13 Mb of memory. As we could expect, the amount of used memory is nearly constant, independently of the number of raised issues (e.g., 250, 500, 750 issues) because the Java garbage collector is able to reclaim unused memory from the JVM heap making it available for subsequent computations. This behaviour of the JVM also explains the higher variance that we can

[2]It is important to consider that having a burst of 1000 issues all together is very rare in real missions.

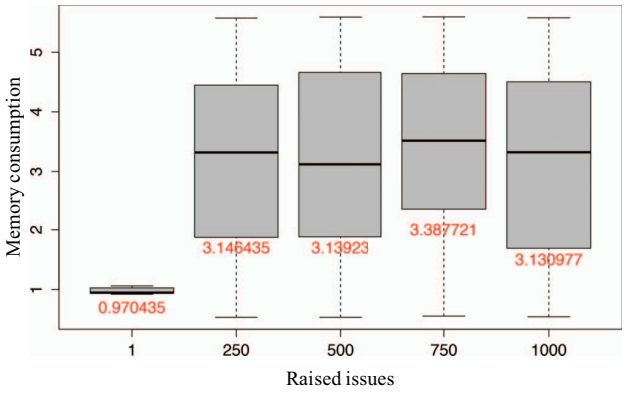

Fig. 6. Memory consumption on Raspberry Pi per number of raised issues (in Mb)

notice when considering the memory consumption during our experiments with respect to the low variance we saw regarding the execution times; indeed, in Java the developer cannot fully control the exact instant in which the garbage collector must be executed, rather the `System.gc()` instruction is a simple *suggestion* to the JVM that does not give any assurance about the exact instant in which the garbage collector is actually executed. Finally, the obtained result about memory usage gives a clear indication about the capability of our implemented CAE to scale for managing real-sized missions. Indeed, we need to recall that many open-source and commercially available UAVs are equipped with far more RAM memory than the one needed in our worst case scenario. For example, the most basic entry-level UAV called AR Drone Parrot is equipped with 1Gb of DDR2 RAM memory. As discussed in Section III, the resolution algorithm is also deployed on much more powerful machines, like ground stations and the central station, which do not suffer from stringent hardware limitations like UAVs do.

Replication package. To allow easy replication and verification of our experiment we provide to interested researchers a complete and portable replication package. The replication package is publicly available[3] and contains all the Java code, R scripts, and obtained data of the experiment.

V. Related Work

In this section we review recent works on coalition formation for multi-agent systems, multi-party sessions and choreographies with utility functions, (hierarchical) component ensembles, and run-time adaptation of multi-robot systems.

Coalition formation has been widely studied in game theory and economics. In multi-agents system, and more in general in Artificial Intelligence (AI), coalition formation has been used as a mean of dynamically creating partnerships or teams of cooperating agents. Many works on coalition formation in multi-agents systems, e.g. [6], [7], use the assumption that all agents can directly communicate with each other, which is not realistic in the real world. [8] tackles the problem of coalition formation in multi-agents systems in a neighbourhood agent network (a network in which agents communicate directly only with their neighbours). Agents can participate to several coalition at the same time, by indicating for each of them the *degree of involvement* (DoI).

Multi agents systems have been applied in variety of applications ranging from the field of electricity markets [9], [10] to supply chains management [11] and complex software systems building [12]. Here, we take in consideration solutions proposed in different areas. Few solutions are application specific, i.e. we regard them as homogeneous agents (ex. [9], [8], [10] etc.) while most of the related work is more general and it includes agents with different types of roles i.e. heterogeneous agents (ex. [13], [14], [15]). All the works include solutions where the agents cooperate to achieve a specific common goal and no study was found which regards the selfish behaviour of the agents. Few studies provide a solution with a centralized coordinator that collects information from all agents and then disseminates a decision to the whole group. However, such a strategy interferes with the system's scalability and robustness: the coordinator can easily become a communication bottleneck, and it is also a potential point of failure for the system. Because of that, most of the works mentioned here propose decentralized approach. For example, in [9] is presented a decentralized and dynamic method where coalition formation is achieved by opportunistic aggregation of agents, while maximizing coalition benefits by means of taking advantage of local resources in the grid.

For what concerns the *utility function*, we have a combination of studies which provide solutions that contain some utility function where agent coalition takes place considering the resources and the performance of the system and studies where solutions don't provide that. For example, in [16] is presented an incremental approach to self-organization based on *bottom-up* coalition formation. Here, agents negotiate to maximize the system's global utility by using a variety of protocols based on local or social marginal utility. Through coalition formation, agents in a large system faced with a set of tasks partition themselves to maximize system performance.

In the literature there are studies that take into consideration the *hierarchy* of the system, thus contributing to scalability. For example, the approach presented in [15] is named Helena - Handling massively distributed systems with ELaborate ENsemble Architectures and it represents a modelling technique centered around the notion of roles teaming up in ensembles. Ensembles are built on top of a component-based platform as goal-oriented communication groups of components. The functionality of each group is described in terms of roles which a component may dynamically adopt. Furthermore, in [17] is described the DEECO (Dependable Emergent Ensembles of Components) component model where the only way components bind and communicate with one another is through ensembles. The basic idea here is a separation of concerns by extracting component bindings and communication from the

[3]http://cs.gssi.infn.it/seaa2016

component implementation and managing them at runtime.

In order to deal with increasingly growing complexity of the mission-critical software systems, there are some recent works which consider *adaptation at runtime*. State-of-the-art mission-critical software systems are often expected to safely adapt to changes in their execution environment where run-time adaptation mechanisms reduce the complexity of the system. For example, Kevoree[4] is an open-source dynamic component model, which relies on models at runtime to properly support the dynamic adaptation of distributed systems. Moreover, in [18] is presented an adaptive run-time model used to establish a flexible information processing within a group of heterogeneous robots, while in [19] is presented a reusable framework for developing adaptive multi-robotic systems for heterogeneous robot teams using an organization based approach. It is based on runtime models for understanding (1) what the system should be doing in terms of system goals and (2) how the system is organized to achieve these goals.

All the work discussed previously focus on one or few of the above mentioned aspects. Contrariwise, our approach is general, meaning that it is not domain or problem specific. This gives us the opportunity to reuse it in different domains. In this work, the approach was adjusted in the domain of UAV-based systems. In contrast to related works, our approach gives the possibility to handle both cooperative and selfish behaviour between agents. Furthermore, the management of coalition is decentralized and this eliminates the single point of failure and the potential bottleneck in the system. Moreover, our approach provides support for a hierarchical structure. This is very useful because it provides scalability and allows different agents with different knowledge to take decision at different levels. Altogether, our approach provides support for run-time adaptation in mission critical and safety critical systems.

VI. Conclusions

In this paper we presented an approach for managing the run-time adaptation of UAV-based systems. The generic adaptation engine that we use identifies the strategy to be used in order to orchestrate the various solvers offered by the entities that compose the UAV-based system. The approach has been empirically evaluated through a large experiment in the context of a private company surveillance scenario.

As future work we are planning to integrate the approach with a suitable extension of the FLYAQ platform [1], [2]. This platform permits to graphically define civilian missions for a team of autonomous multicopters via a domain specific language especially conceive to "democratize" the use of such drones, i.e. to make the specification of missions accessible to people with no expertise in IT and robotics (some preliminary results of this line of research have been presented in [20]). Moreover, we plan to further experiment the approach in different domains and in real environments.

[4]http://kevoree.org/

VII. Acknowledgements

This work is partially funded by the 7th Framework EU-FET project 600792 ALLOW Ensembles.

References

[1] D. Di Ruscio, I. Malavolta, and P. Pelliccione, "Engineering a platform for mission planning of autonomous and resilient quadrotors," in *SERENE 2013, Kiev, Ukraine, October 3-4, 2013. Proceedings*, 2013, pp. 33–47. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40894-6_3

[2] D. Bozhinoski, D. Di Ruscio, I. Malavolta, P. Pelliccione, and M. Tivoli, "FLYAQ: Enabling Non-Expert Users to Specify and Generate Missions of Autonomous Multicopters," in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, 2015.

[3] T. L. Saaty, *What is the analytic hierarchy process?* Springer, 1988.

[4] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner, "Control flow requirements for automated service composition," in *ICWS 2009*, 2009, pp. 17–24.

[5] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*, ser. Computer Science. Springer, 2012.

[6] T. P. Michalak, J. Sroka, T. Rahwan, M. Wooldridge, P. McBurney, and N. R. Jennings, "A distributed algorithm for anytime coalition structure generation." in *AAMAS*, W. van der Hoek, G. A. Kaminka, Y. Lesprance, M. Luck, and S. Sen, Eds. IFAAMAS, 2010, pp. 1007–1014.

[7] S. D. Ramchurn, M. Polukarov, A. Farinelli, C. Truong, and N. R. Jennings, "Coalition formation with spatial and temporal constraints." in *AAMAS*, W. van der Hoek, G. A. Kaminka, Y. Lesprance, M. Luck, and S. Sen, Eds. IFAAMAS, 2010, pp. 1181–1188.

[8] D. Ye, M. Zhang, and D. Sutanto, "Self-adaptation-based dynamic coalition formation in a distributed agent network: A mechanism and a brief survey," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 5, pp. 1042–1051, 2013.

[9] R. Mihailescu, M. Vasirani, and S. Ossowski, "Dynamic coalition adaptation for efficient agent-based virtual power plants," in *Multiagent System Technologies - 9th German Conference, MATES 2011, Berlin, Germany, October 6-7, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6973. Springer, 2011, pp. 101–112.

[10] T. Pinto, H. Morais, P. Oliveira, Z. Vale, . Praa, and C. Ramos, "A new approach for multi-agent coalition formation and management in the scope of electricity markets," *Energy*, vol. 36, no. 8, pp. 5004–5015, 2011.

[11] H. J. Ahn, H. Lee, and S. J. Park, "A flexible agent system for change adaptation in supply chains," *Expert Syst. Appl.*, vol. 25, no. 4, pp. 603–618, 2003.

[12] N. R. Jennings, "An agent-based approach for building complex software systems," *Commun. ACM*, vol. 44, no. 4, pp. 35–41, 2001.

[13] M. D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro, "Developing correct, distributed, adaptive software," *Sci. Comput. Program.*, vol. 97, pp. 41–46, 2015.

[14] R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "A formal approach to autonomic systems programming: The SCEL language," *TAAS*, vol. 9, no. 2, p. 7, 2014.

[15] R. Hennicker and A. Klarl, "Foundations for ensemble modeling - the helena approach - handling massively distributed systems with elaborate ensemble architectures," in *Specification, Algebra, and Software*, 2014, pp. 359–381.

[16] M. Sims, C. V. Goldman, and V. R. Lesser, "Self-organization through bottom-up coalition formation." in *AAMAS*, 2003, pp. 867–874.

[17] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil, "Deeco: an ensemble-based component system," in *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*. ACM, 2013, pp. 81–90.

[18] S. Niemczyk and K. Geihs, "Adaptive run-time models for groups of autonomous robots," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*. IEEE, 2015, pp. 127–133.

[19] C. Zhong and S. A. DeLoach, "Runtime models for automatic reorganization of multi-robot systems," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 20–29.

[20] D. Bozhinoski, I. Malavolta, A. Bucchiarone, and A. Marconi, "Sustainable Safety in Mobile Multi-robot Systems via Collective Adaptation," in *Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th International Conference on*. IEEE, 2015, pp. 172–173.