ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

Fostering resilient execution of multi-agent plans through self-organisation

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Audrito G., Casadei R., Torta G. (2021). Fostering resilient execution of multi-agent plans through self-organisation. Institute of Electrical and Electronics Engineers Inc. [10.1109/ACSOS-C52956.2021.00076].

Availability:
This version is available at: https://hdl.handle.net/11585/876121 since: 2022-03-01

Published:
DOI: http://doi.org/10.1109/ACSOS-C52956.2021.00076

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is the final peer-reviewed accepted manuscript of:

G. Audrito, R. Casadei and G. Torta, "Fostering resilient execution of multi-agent plans through self-organisation," 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), DC, USA, 2021, pp. 81-86

The final published version is available online at ***https://dx.doi.org/10.1109/ACSOS-C52956.2021.00076***

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

Fostering resilient execution of multi-agent plans through self-organisation

Giorgio Audrito
Dipartimento di Informatica
Università di Torino
Torino, Italy
0000-0002-2319-0375

Roberto Casadei
Department of Computer Science and Engineering
Università di Bologna
Bologna, Italy
0000-0001-9149-949X

Gianluca Torta
Dipartimento di Informatica
Università di Torino
Torino, Italy
0000-0002-4276-7213

***Abstract*—Traditional multi-agent planning addresses the coordination of multiple agents towards common goals, by producing an integrated plan of actions for each of those agents. For systems made of large numbers of cooperating agents, however, the execution and monitoring of a plan should enhance its high-level steps, possibly involving entire sub-teams, with a flexible and adaptable lower-level behaviour of the individual agents. In order to achieve such a goal, we need to integrate the behaviour dictated by a multi-agent plan with self-organizing, swarm-based approaches, capable of automatically adapting their behaviour based on the contingent situation, departing from the predetermined plan whenever needed. Moreover, in order to deal with multiple domains and unpredictable situations, the system should, as far as possible, exhibit such capabilities without hard-coding the agents behaviour and interactions. In this paper, we investigate the relationship between multi-agent planning and self-organisation through the combination of two representative approaches both enjoying declarativity. We consider a functional approach to self-organising systems development, called Aggregate Programming (AP), and propose to exploit collective adaptive behaviour to carry out plan revisions. We describe preliminary results in this direction on a case study of execution monitoring and repair of a Multi-Agent PDDL plan.**

***Index Terms*—Multi-agent systems, Robust plan execution, Aggregate computing**

I. Introduction

The task of Multi-Agent Planning (MAP) consists of coordinating the actions of multiple agents in a Multi-Agent System (MAS) towards common goals. Traditionally, planning addresses both the assignment of tasks/actions to individual agents, as well as the coordination among different agents in terms of *causal links* and *concurrent* actions [1]. For large-scale systems of cooperating agents, however, plans can hardly capture, fully and in advance, the behaviour and interactions of each individual agent towards desired global state-of-affairs. Rather, *high-level* plans should be specified and dynamically refined to define both team- and individual-level behaviour in a flexible and adaptable fashion.

The vision presented in this paper aims at achieving such a goal by combining and integrating two approaches to MAS implementation that have followed distinct research paths up to now. On the one hand, there are self-organizing, swarm-based approaches, capable of automatically adapting MAS behaviour based on the contingent situation, departing from the predetermined behaviour whenever needed. Such approaches can be very effective in dealing with small, specific uncertainties in the operating environment, but are usually not suited to the execution of complex plans made of several phases, in which groups of agents and individual agents must be able to exhibit very different behaviours and coordination capabilities. On the other hand, there are more traditional systems that are able (at least in principle) to interpret and execute any plan expressed in a suitably standardized language, such as the Multi-Agent-Planning Domain Definition Language (MA-PDDL) [2], but do not have built-in capabilities to exhibit flexible lower-level behaviour and coordination with other agents.

In other words, we aim at investigating the integration between multi-agent plans and self-organisation, where the former are necessary to achieve complex goals that require possibly long sequences of agents' actions; and the latter are necessary to deal with the uncertainties and complexities of the execution environment through low-level flexibility.

A fundamental requirement of our investigation is that both of these ingredients should be based as far as possible on *declarative* solutions: the high-level plans should not be hard-coded in the system, but represented in a suitable language; and also the self-organization and flexible behaviour capabilities of the agents should be expressible by programming at a high level, abstracting from low-level details such as inter-agent communications. For the latter goal, we consider a functional approach to self-organising systems development, called Aggregate Programming (AP) [3], that makes it possible to express collective adaptive behaviour to carry out plan execution and revision. As we shall review, AP is formalised through the Field Calculus (FC) [4] and implemented by full programming languages such as ScaFi [5].

We make a preliminary test of our vision on a case study of execution monitoring and repair of a MA-PDDL plan. In particular, we present the architecture of an actor-based (simulated) system for joint execution of MAP and AP programs. The system takes a MA-PDDL plan and simulates its execution, possibly with failures preventing the achievement of the plan goals. As the plan is executed, an AP program written in ScaFi (collectively run by the actors, as we shall see) monitors the status of the system, and in case of failure triggers a distributed repair process involving not only the agents executing the plan, but possibly also additional nodes

in the geographic environment where the MAP is executed.

The paper is organised as follows. Section II provides background on MAP, AP and self-organization hinting at the gaps that motivate our proposal. Section III describes a preliminary investigation of combining MAP with self-organisation. Finally, Section IV wraps up the paper with a discussion and perspectives for further research.

II. Background and Motivation

A. Multi-Agent Planning (MAP)

MAP evolves from one of the oldest AI problems: automated planning of the actions that an agent has to execute in order to reach a *goal* state from starting in an *initial* state. In 2012, the standard language for expressing planning domains and problems (PDDL) has been extended to the MA-PDDL language, which can handle multiple agents [2]. Currently, several planners directly support MAP [6], both as a centralized and as a decentralized process. As an alternative to using such planners, it is possible to automatically convert a MA-PDDL problem to a single-agent problem [7] that can then be solved with one of the many single-agent planners.

A potential issue with the execution of a MA-PDDL plan is that, if errors occur, it is left unspecified how the single agents and the MAS as a whole should react. Some work has been done in the AI community to address monitoring and repair of Multi-Agent Plans (MAPs) [8]–[11], however, such approaches only address the monitoring/diagnosis task, assume a centralized monitoring/repair process, or require full/perfect communication between the agents in the team.

It is worth mentioning that the multi-agent community has also followed a somewhat different approach to the MAP problem. More specifically, it has identified a number of problem types that are particularly relevant in practice, such as the *path-planning* problem, the *covering* problem (i.e., observe a set of areas) and the *pickup-and-delivery* problem (i.e., move a set of items from sources to destinations). Such problem types have then been investigated separately, leading to specialized solutions that are often partially hardcoded in the agents behaviours [12], [13].

The main shortcoming of the mentioned approaches is that they usually lack the capability to capture flexible, reactive collective behaviour of agents in the presence of uncertainty and unforeseen events. This is true, even if to a lesser extent, also for the solutions to specific types of problems.

B. Aggregate Programming (AP) and the Field Calculus (FC)

AP [3], [14] is an approach for designing resilient distributed systems by abstracting away from individual devices behaviour and focusing on the global, aggregate behaviour of the collection of all devices. The AP approach provides smooth composition of distributed adaptive behaviour, allowing for the development of highly reusable "building block" operators capturing common coordination patterns [15], [16]. Most importantly, AP assumes only local communication between neighbour agents, and is robust with respect to devices joining/leaving the network, or failing.

AP is formally backed by FC [3], [4], tiny functional language for expressing aggregate programs (see [4] for a detailed account), which is implemented by full-fledged languages like the Scala-internal *ScaFi (Scala Fields)* [5] and the C++ internal *FCPP* [17]. A summary of the syntax of FC is as follows:

$\mathtt{P} ::= \overline{\mathtt{F}}\ \mathtt{e}$		program
$\mathtt{F} ::= \mathtt{def}\ \mathtt{d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$		function declaration
$\mathtt{e} ::= \mathtt{x} \mid \mathtt{v} \mid (\overline{\mathtt{x}})\mathtt{=>e} \mid \mathtt{e}(\overline{\mathtt{e}}) \mid$		expression
$\mathtt{let}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e} \mid \mathtt{if}(\mathtt{e})\{\mathtt{e}\}\{\mathtt{e}\} \mid$		
$\mathtt{rep}(\mathtt{e})\{(\mathtt{x})\mathtt{=>e}\} \mid \mathtt{nbr}\{\mathtt{e}\} \mid \mathtt{spawn}(\mathtt{e},\mathtt{e},\mathtt{e})$		

In FC, the main expression $\mathtt{e}$ evaluates to a possibly different value $\mathtt{v}$ on each device δ, which may depend on the state of δ (sensor readings, information received from neighbours, etc.). Thus, expressions $\mathtt{e}$ induce a *computational field* Φ, which can be represented as a time-varying map $\delta_1 \mapsto \mathtt{v}_1, \ldots, \delta_n \mapsto \mathtt{v}_n$, assigning a value $\mathtt{v}_i$ to each device δ_i in a network. Each device δ updates its value (by evaluating $\mathtt{e}$) in asynchronous computational rounds, and values can be either *local* (of any recursive data type), or *neighbouring values* ϕ (maps from neighbour devices δ' to local values). Besides standard constructs of functional languages (function declaration, let-binding, function call, anonymous functions and branching), the syntax of expressions comprises three peculiar constructs:

- $\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\mathtt{=>e}_2\}$ (state evolution), which on a device δ evaluates $\mathtt{e}_2$ substituting $\mathtt{x}$ with the value calculated for the whole $\mathtt{rep}$-expression at the previous round on δ (in the first round, with the value of $\mathtt{e}_1$);
- $\mathtt{nbr}\{\mathtt{e}\}$ (neighbourhood observation), which produces a neighbouring value representing an "observation map" of neighbour's values for expression $\mathtt{e}$, i.e., a map from neighbours to their latest evaluation of $\mathtt{e}$.
- $\mathtt{spawn}(\mathtt{e}_0, \mathtt{e}_1, \mathtt{e}_2)$ (dynamic process generation and execution [18]), spawning an *aggregate process* corresponding to $\mathtt{e}_0$ for every *key* contained in the set given by $\mathtt{e}_1$, passing the *value* of $\mathtt{e}_2$ as further input to each of them. The aggregate process $\mathtt{e}_0$ must be a function taking as argument a key and a value, and returning a pair of a result and a *status*, which may be either: *OUTPUT* (make the result available to the calling FC program), *BUBBLE* (no result returned, device takes part in the process), *EXTERNAL* (the device does not participate in the process this round), *TERMINATED* (the whole process should terminate). Once spawned, a process expands from devices in *BUBBLE* or *OUTPUT* status to corresponding neighbour nodes, until it somewhere returns *TERMINATED*. We shall clarify $\mathtt{spawn}$ through examples in Section III-C (see [18] for a detailed account).

As syntactic sugar, in this paper we use binary operators in infix notation and the notation $[\mathtt{e}_1, \ldots, \mathtt{e}_n]$ for tuples. We shall use several built-in functions, whose meaning will be mostly self-explanatory. Among them, we shall use the multiplexer built-in function $\mathtt{mux}(\mathtt{e}_0, \mathtt{e}_1, \mathtt{e}_2)$, which returns its second or third argument depending on the Boolean value of the first, and the $\mathtt{foldhood}(\mathtt{e}_0, \mathtt{e}_1, \mathtt{e}_2)$ function which aggregates

neighbouring field e_2 together with a starting value e_0 and a binary aggregation operator e_1.

C. Multi-Agent Planning and Self-Organisation

Self-organisation refers to the process whereby a system autonomously (i.e., without external control) seeks and sustains its order or structures [19]. It is often meant as a bottom-up decentralised process where macro-level structures *emerge* from micro-level activities and interactions. Engineering self-organising software-based systems [20] typically leverages mechanisms inspired by nature (e.g., chemistry, physics, animals, insects). However, few *programming* approaches tailored to self-organising systems exist, with AP being the most developed [3], to the best of our knowledge.

Our vision is that self-organisation can be profitable combined with complementary approaches to multi-agent coordination. The general idea itself is not novel: for instance, research on *organic computing* proposes to balance "creative self-organized bottom-up processes" and "top-down control" [21]. Research on *organisational paradigms* for MASs [22] also distinguishes between problem-driven organisations achieved by task planning, which have issues in dynamic environments, and self-organising organisational approaches, which can better deal with continuous perturbations and unpredictable situations but are usually limited in scope. In [23], a comprehensive view of organisations in MASs is depicted through two dimensions: organisational awareness (aware, unaware) and centrality (agent-centred vs. organisation-centred)— where self-organisation is classified as agent-centred and organisation-unaware.

Our contribution has intersections with the aforementioned threads, but also a distinct focus: supporting the execution of MAPs, i.e., possibly long and complex sequences of actions that have to be performed by the MAS. In particular:

- we consider the *plan* (together with declarative models of the actions) as the main force used to drive the (dynamic) structure of a self-organising MAS, by defining workflows and actions to "steer" the self-organisation;
- we advocate that MAPs *declarative representations* should be developed that comprise both collective actions performed by teams of agents, as well as actions to be performed by individual agents, as required by PDDL extensions;
- we consider the full execution cycle (including monitoring and repair of the plans), which requires both robust and flexible plan execution, and incremental (partial-to-full) re-planning when strictly necessary.

We provide preliminary results in the following section. In particular, self-organisation can be a key mechanism for addressing *continual* planning [24]. An architectural solution leveraging planning and self-organisation in robotic ensembles has also been proposed in [25]. Their goal has some similarities with ours, however they ignore the fundamental role that a suitable representation of the plan can have in automatically generating the code for monitoring its correct execution and possible repair. Our vision, instead, implies a strong focus on a declarative representation of the plan (especially the action models) as well as of the monitoring and repair processes (c.f. Section IV). We believe that declarativity is essential for decoupling the system specification from execution and deployment issues, as well as for enabling formal analysis including static and dynamic verification.

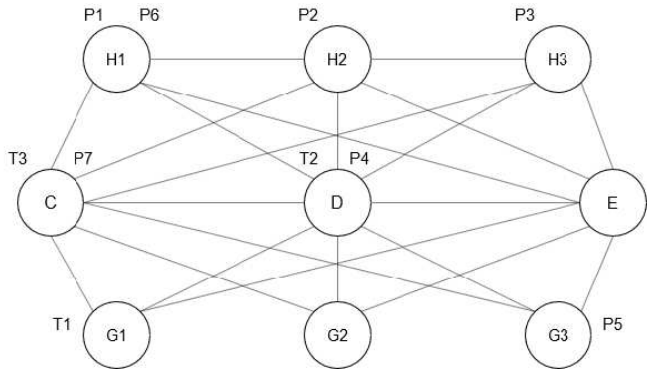

Fig. 1. A sample map for the taxi domain.

```
(:action drive
   :agent    ?t - taxi
   :parameters  (?from - location ?to - location)
   :precondition  (and  (at ?t ?from)
       (directly-connected ?from ?to)
       (free ?to)
     )
   :effect    (and (not (at ?t ?from))
       (not (free ?to))
       (at ?t ?to)
       (free ?from)
     )
)
```

Fig. 2. The *drive* action for the *Taxi* PDDL domain.

III. Preliminary Investigation: Plan Repair

A. Domain: Taxis and Passengers

We take the *taxi* MA-PDDL domain from the CoDMAP competition [26] as our running example, which defines two types of agents: taxis and passengers. Taxis can move from one location to another in a graph-based map, can carry one passenger at a time, and have the goal to reach a final location at the end of the execution. Passengers can enter and leave taxis, and have the goal to reach a destination at the end of the execution. The domain is quite simple, but exhibits a basic form of collaboration among agents: a passenger cannot reach her destination without the help of a taxi. Figure 1 shows an example problem with 9 locations (labeled circles), 3 taxis ($T1, \ldots, T3$) and 7 passengers ($P1, \ldots, P7$).

In our prototype, we automatically translate a MA-PDDL problem to PDDL 2.1 [27], which allows durative actions, while the currently available MA-PDDL planners only allow for atomic actions. Plans to be fed into our simulator have then been generated with the POPF2 planner [28].

Figure 2 shows the definition of the *drive* action within the PDDL specification for the Taxi domain. What is important to

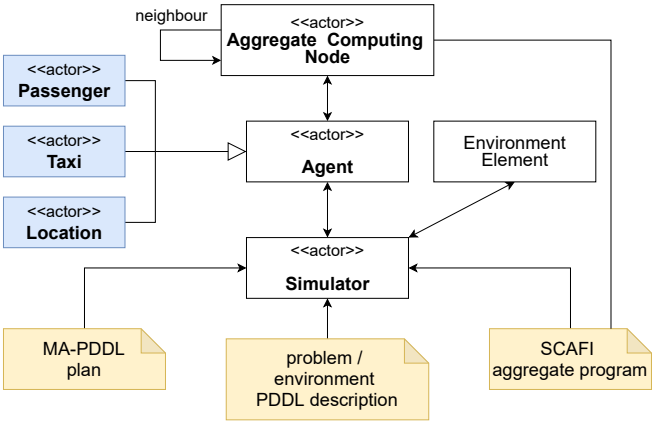

Fig. 3. Actor-based architecture of the system. Inputs are shown in yellow, domain-specific entities are shown in blue.

note for the purpose of our paper, is that such a definition constitutes a declarative model of the action, with pre-conditions and effects on the environment. Such a model can be exploited not only for the generation of a plan, but also for simulating and monitoring the plan execution. We can think of a plan, together with the action models, as a formal specification of the runtime behaviour of the MAS that executes it.

B. Actor-based System Architecture

In this section, we describe the architecture of a system for the execution and repair of MA-PDDL multi-agent plans. The proposed system simulates in software the execution of actions by the agents, and does not address specific hardware agents. However, it is intended to model quite closely a possible software architecture of a system deployed on real software/hardware agents.

The overall architecture is based on actors that implement both the agents of the MAS and the nodes of a network for the execution of aggregate programs [29]. Figure 3 shows the architecture of the simulation system, including the domain-specific actors (cf. Section III-A) involved in the execution of a simple plan. Generic entities of the solution (implemented in Scala exploiting the AKKA framework [30]) include:

1) *(discrete-event) simulator*:
   a) creates a *future event list (FEL)* with start/end action events from the plan **and** failure events from the simulation configuration, sending corresponding messages to agent-actors;
   b) implements the communication and sensor infrastructure needed by FC, i.e., sending to the node-actors current sensor values and *context*;
2) *agent* actors: model active domain entities and have associated aggregate computing nodes;
3) *aggregate computing nodes*:
   a) have a FC/ScaFi *aggregate program* to run, a *state* (output of their last execution round), a set of *neighbours* and their most recent *export* (i.e., a collection of values produced by *nbr* and *rep* constructs to be shared with neighbours [4]);
   b) can perform actions `get-context` (retrieving the current state, neighbours' exports, and sensor data), `compute` (running the aggregate program against the current context), and `act` (performing actuations).

Domain-specific entities include the following:

1) *taxi* agent actors:
   a) have a `location` and may be serving one passenger;
   b) can perform action `drive` (for moving a taxi and the corresponding passenger, if any, from a location to another one);
2) *passenger* agent actors:
   a) have a `location`, a target destination, and may be in a taxi;
   b) can perform actions `enter` (to get into a taxi), `exit` (to get off a taxi);
3) *location* agent actors:
   a) have a set of connected locations that can be directly reached by a taxi performing a `drive` action;
   b) are assumed to host a computational element, providing storage and computing services for a given location.

All the entities (taxis, passengers, and locations) are also nodes of an aggregate computing system, where we assume that passengers have a smartphone or another wearable device for computation, and locations have an associated smart-city device. Devices corresponding to taxis and passengers in a same location are all connected together and with the smart-city device of the location. Additionally, location devices are also connected to the devices of neighbour locations.

The simulator exploits the plan and the problem description to determine a list of events (FEL) driving the simulation. For each event, a corresponding message is sent to the involved actors, which perform planned actions and reply back with results so that new events can be scheduled affecting the environment or other entities.

C. Aggregate Plan Repair

Monitoring and repair have been implemented in the AP nodes with the Scala-internal *ScaFi* language [5]. However, for ease of explanation, in this section we shall describe the repair algorithm using the syntax of FC described in section II. The FC program that monitors the plan execution and triggers a distributed repair when an error occurs is shown in Figure 4. We highlight keywords in blue, general built-in functions in violet, values and functions accessing plan and model information in red, comments in green. For simplicity, we assume that the only error that can happen is that a taxi breaks and can no longer serve its assigned passengers.

Firstly, FC nodes keep an up-to-date set *brokenTaxis* of currently broken taxis. Taxi nodes whose *taxiBroken* sensor

```
let brokenTaxis = rep (emptyset()) { old => foldhood(if (taxiBroken) {singleton(mid)} {emptyset()}, union, nbr{old}) } in
let repairNeeded = !hasTaxi and contains(brokenTaxis, myTaxi) in
// INPUT: key = [passenger, location, location], exec  OUTPUT: [[time, taxi], STATUS]
let evaluateTaxi = (key, exec) => {
  let myOffer = mux(isTaxi, tuple(timeForLocationAfterPlan(2nd(key)) + 1st(route(2nd(key), 3rd(key))), mid), tuple(infinity, null)) in
  let status = mux(1st(key) == mid, mux(exec, OUTPUT, TERMINATED), BUBBLE) in
     tuple(gossipMin(myOffer), status)
} in
let fallbackTaxi = rep (none) { old =>
  let callProcess = (old == none) and repairNeeded in
  let spawnKeys = mux(callProcess, singleton(tuple(mid, curLoc, trgLoc)), emptyset()) in
  let bestTaxi = get(spawn(evaluateTaxi,  spawnKeys, callProcess), mid) in
  let timeSinceImprove = 1st(rep (tuple(0, infinity)) { old =>
    mux(bestTaxi == none,
        tuple(0, infinity),
        tuple(mux(1st(bestTaxi) < 2nd(old), 0, 1st(old) + 1), 1st(bestTaxi)) )
  } in mux(timeSinceImprove > THRESHOLD, 2nd(bestTaxi), old)
} in
// INPUT: key = [passenger, taxi, location, location] args = unit OUTPUT: [unit, STATUS]
let callTaxi = (key, args) => { mux(2nd(key) == mid, (addToPlan(1st(key), 3rd(key), 4th(key)); TERMINATED), BUBBLE) } in
let spawnKeys = mux(fallbackTaxi == none, emptyset(), singleton(tuple(mid, fallbackTaxi, curLoc, trgLoc))) in
spawn(callTaxi, spawnKeys, unit()) // spawns a process instance with logic 'callTaxi' for each (new) key in 'spawnKeys'
```

Fig. 4. Field Calculus code for handling broken taxis.

indicates a failure increase this set by adding their unique node ID *mid* to the set. At the same time, each passenger node determines whether she needs a repair of her plan (*repairNeeded*), which happens when she is waiting for a taxi (*hasTaxi*) that is in the *brokenTaxis* set.

When a passenger node needs a plan repair, it tries to find a *fallbackTaxi* by spawning a FC process based on function *evaluateTaxi*. Let us consider in some detail the *rep* expression used to set variable *fallbackTaxi*. Flag *callProcess* is set to *true* iff in the previous rounds no fallback taxi had been identified (*old* variable is *none*) and the code is evaluated on a passenger node with *repairNeeded* equal to *true*. When *callProcess* is *true*, the key (identifier) *spawnKeys* of the spawned process is the singleton set of a 3-tuple containing the ID *mid* of the node and its current and target locations.

A new process based on function *evaluateTaxi* is then spawned. Note that processes are spawned only at the nodes of passengers needing a plan repair. However, the semantics of *spawn* is such that, after the generation of a process, at each round the active partecipants propagate it to their neighbours. The process function *evaluateTaxi* computes a value (a pair of a time and the ID of the suggested taxi node) and a status; when executed on the passenger node that spawned the process, the status is either *OUTPUT* or *TERMINATED*, based on parameter *exec* which is set with the value of flag *callProcess*; for all other nodes, the status is *BUBBLE*. The computed value (which is returned only with the *OUTPUT* status) is determined through an aggregate computation involving all the nodes, and contains the estimated time that will take to the suggested taxi to serve the passenger. Since the spawned process takes several rounds before converging to a final value with an (estimated) optimal suggestion, it is kept alive until it does not improve its output for more than a *THRESHOLD* number of rounds. When that happens, the taxi suggested by *evaluateTaxi* becomes the chosen *fallbackTaxi* for the passenger, causing the process to terminate.

Finally, the passenger must let the chosen taxi know her decision. This task is also performed by spawning a process, this time based on the *callTaxi* function. The process is spawned by the passenger nodes and, when its propagation reaches the *fallbackTaxi* node, *callTaxi* updates the MAP according to the chosen fix.

IV. Discussion and Research Roadmap

The work described in the previous section is a preliminary step in integrating the execution of a MAP with the collective behaviour generated by AP. These early results have several important limitations, most notably:

- the *plan* is expressed in MA-PDDL, which is not suitable to express collective, high-level actions such as those needed when (possibly large) teams of agents carry plan actions that often involve entire sub-teams;
- consequently, the *execution* of the plan does not exhibit any flexibility (e.g. how many and which agents should perform a collective action?), unless a failure occurs;
- the *monitoring* assumes full observability of the relevant information needed to detect failures (that may not be true especially for the outcomes of aggregate actions);
- the *repair* of the plan is a hard-coded AP process written for a specific domain and kind of failure.

The list above shows that there is still a long way to achieve our vision. However, we already have some ideas about the needed steps and techniques that may be helpful.

- A fundamental step would be the development of a language for expressing MAPs. A first step in this direction has been presented in [31], where the authors propose the notion of *aggregate plan* to capture the kind of plans suitable for teams performing collective actions. We still need to formalize a language to express aggregate plans, and the models of the actions involved (in terms, e.g., of pre-conditions, nominal and faulty post-conditions).
- Given an aggregate plan, its execution must be flexible enough. This will require to define a layer exploiting the

expressive power of FC to induce such flexible collective behavior in a fully distributed, self-organizing way.

- The properties to be monitored may require more complex mechanisms than just direct observation by individual agents, such as those investigated in existing work on Runtime Monitoring of complex spatial and temporal properties with FC [32]. Furthermore, these properties should be automatically derived from the plan and the actions model, instead of manually specified.
- Finally, we envision a layer for the repair of failures that goes beyond the flexibility directly exhibited by the execution layer. The characteristics of FC and of the systems we address seem to suggest that also such a layer should be an FC process. In order to avoid hardcoding the repair actions, the layer should exploit knowledge of the plan and of the actions (failure) models in order to update the plan itself. In this way, we will probably start by diagnosing (i.e. assessing) the situation (e.g., several observed delays are due to a congestion) and subsequently finding the minimal plan change required to put the execution back on track.

The practical long-term goal of our proposal is to achieve a unified approach to diverse applications that are currently addressed by specific solutions that are either swarm-based (e.g., search-and-rescue, crowd safety) or plan-based (e.g. pickup-and-delivery, area covering, etc.). This should open the way to address hybrid scenarios requiring the execution of complex plans that involve, at least partially, swarms of agents. We believe that a declarative, high-level specification of the goals, actions and single agents behaviors will help making the implementation of such systems feasible.

References

[1] C. Boutilier and R. I. Brafman, "Partial-order planning with concurrent interacting actions," *Journal of Artificial Intelligence Research*, vol. 14, pp. 105–136, 2001.

[2] D. L. Kovács, "A multi-agent extension of PDDL3.1," in *International Planning Competition (IPC)*, 2012.

[3] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019.

[4] G. Audrito, M. Viroli, F. Damiani, D. Pianini, and J. Beal, "A higher-order calculus of computational fields," *ACM Trans. Comput. Log.*, vol. 20, no. 1, pp. 5:1–5:55, 2019.

[5] R. Casadei, M. Viroli, G. Audrito, and F. Damiani, "Fscafi : A core calculus for collective adaptive systems programming," in *ISoLA (2)*, ser. Lecture Notes in Computer Science, vol. 12477. Springer, 2020, pp. 344–360.

[6] A. Torreno, E. Onaindia, A. Komenda, and M. Štolba, "Cooperative multi-agent planning: a survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–32, 2017.

[7] M. Crosby and R. Petrick, "Temporal multiagent planning with concurrent action constraints," in *ICAPS workshop on distributed and multi-agent planning (DMAP)*, 2014.

[8] F. de Jonge, N. Roos, and C. Witteveen, "Primary and secondary diagnosis of multi-agent plan execution," *Journal of Autonomous Agent and Multiagent Systems*, vol. 18, pp. 267–294, 2009.

[9] M. Kalech, "Diagnosis of coordination failures: A matrix-based approach," *Journal of Autonomous Agents and Multiagent Systems*, vol. 24, no. 1, pp. 69–103, 2012.

[10] R. Micalizio, "Action failure recovery via model-based diagnosis and conformant planning," *Computational Intelligence*, vol. 29, no. 2, pp. 233–280, 2013.

[11] R. Micalizio and P. Torasso, "Cooperative Monitoring to Diagnose Multiagent Plans," *Journal of Artificial Intelligence Research*, vol. 51, pp. 1–70, 2014.

[12] M. Liu, H. Ma, J. Li, and S. Koenig, "Task and path planning for multi-agent pickup and delivery," in *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2019.

[13] O. Salzman and R. Stern, "Research challenges and opportunities in multi-agent path finding and multi-agent pickup and delivery problems," in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, 2020, pp. 1711–1715.

[14] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the Internet of Things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015.

[15] M. Viroli, G. Audrito, J. Beal, F. Damiani, and D. Pianini, "Engineering resilient collective adaptive systems by self-stabilisation," *ACM Transactions on Modelling and Computer Simulation*, vol. 28, no. 2, pp. 16:1–16:28, 2018.

[16] R. Casadei, D. Pianini, M. Viroli, and A. Natali, "Self-organising coordination regions: A pattern for edge computing," in *COORDINATION*, ser. Lecture Notes in Computer Science, vol. 11533. Springer, 2019, pp. 182–199.

[17] G. Audrito, "FCPP: an efficient and extensible field calculus framework," in *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 153–159.

[18] R. Casadei, M. Viroli, G. Audrito, D. Pianini, and F. Damiani, "Aggregate processes in field calculus," in *International Conference on Coordination Languages and Models*. Springer, 2019, pp. 200–217.

[19] T. De Wolf and T. Holvoet, "Emergence versus self-organisation: Different concepts but promising when combined," in *Engineering Self-Organising Systems, Methodologies and Applications (ESOA workshop, AAMAS conference)*, ser. Lecture Notes in Computer Science, S. Brueckner, G. D. M. Serugendo, A. Karageorgos, and R. Nagpal, Eds., vol. 3464. Springer, 2004, pp. 1–15.

[20] H. V. D. Parunak and S. A. Brueckner, "Software engineering for self-organizing systems," *Knowl. Eng. Rev.*, vol. 30, no. 4, pp. 419–434, 2015.

[21] J. Branke, M. Mnif, C. Müller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck, "Organic computing - addressing complexity by controlled self-organization," in *Leveraging Applications of Formal Methods (ISoLA)*. IEEE Computer Society, 2006, pp. 185–191.

[22] C. J. Amaral and J. F. Hübner, "From goals to organisations: Automated organisation generator for MAS," in *Engineering Multi-Agent Systems (EMAS)*, ser. Lecture Notes in Computer Science, vol. 12058. Springer, 2019, pp. 25–42.

[23] G. Picard, J. F. Hübner, O. Boissier, and M.-P. Gleizes, "Reorganisation and self-organisation in multi-agent systems," in *1st International Workshop on Organizational Modeling, ORGMOD*, 2009, pp. 66–80.

[24] M. Brenner and B. Nebel, "Continual planning and acting in dynamic multiagent environments," *Autonomous Agents and Multi Agent Systems*, vol. 19, no. 3, pp. 297–331, 2009.

[25] O. Kosak, C. Wanninger, A. Hoffmann, H. Ponsar, and W. Reif, "Multipotent systems: Combining planning, self-organization, and reconfiguration in modular robot ensembles," *Sensors*, vol. 19, no. 1, p. 17, 2019.

[26] "Competition of Distributed and Multiagent Planners (CoDMAP)," http://agents.fel.cvut.cz/codmap, accessed: 2021-02-12.

[27] M. Fox and D. Long, "PDDL2. 1: An extension to PDDL for expressing temporal planning domains," *Journal of artificial intelligence research*, vol. 20, pp. 61–124, 2003.

[28] A. Coles, A. Coles, M. Fox, and D. Long, "POPF2: A forward-chaining partial order planner," *The 2011 International Planning Competition*, vol. 65, 2011.

[29] R. Casadei and M. Viroli, "Programming actor-based collective adaptive systems," in *Programming with Actors*. Springer, 2018, pp. 94–122.

[30] "Build powerful reactive, concurrent, and distributed applications in Java and Scala (AKKA)," https://akka.io, accessed: 2021-02-12.

[31] M. Viroli, D. Pianini, A. Ricci, and A. Croatti, "Aggregate plans for multiagent systems," *International Journal of Agent-Oriented Software Engineering*, vol. 5, no. 4, pp. 336–365, 2017.

[32] G. Audrito, R. Casadei, F. Damiani, V. Stolz, and M. Viroli, "Adaptive distributed monitors of spatial properties for cyber-physical systems," *Journal of Systems and Software*, vol. 175, 2021.