Multi-agent Systems Meet Aggregate Programming: Towards a Notion of Aggregate Plan

Mirko Viroli(✉), Danilo Pianini, Alessandro Ricci, Pietro Brunetti, and Angelo Croatti

University of Bologna, Bologna, Italy
{mirko.viroli,danilo.pianini,a.ricci,p.brunetti,a.croatti}@unibo.it

Abstract. Recent works foster the idea of engineering distributed situated systems by taking an aggregate stance: design and development are better conducted by abstracting away from individuals' details, directly programming overall system behaviour instead. Concerns like interaction protocols, self-organisation, adaptation, and large-scaleness, are automatically hidden under the hood of the platform supporting aggregate programming. This paper aims at bridging the apparently significant gap between this idea and agent autonomy, paving the way towards an aggregate computing approach for multi-agent systems. Specifically, we introduce and analyse the idea of "aggregate plan": a collective plan to be played by a dynamic team of cooperating agents.

1 Introduction

Self-organisation mechanisms support adaptivity and resilience in complex natural systems at all levels, from molecules and cells to animals, species, and entire ecosystems. A long-standing aim in computer science is to find effective engineering methods for exploiting such mechanisms to bring similar adaptivity and resilience to a wide variety of complex, large-scale applications—in smart mobility, crowd engineering, swarm robotics, etc. Practical adoption, however, poses serious challenges, since self-organisation mechanisms often trade efficiency for resilience and are often difficult to predictably compose to meet more complex specifications.

On the one hand, in the context of multi-agent systems, self-organisation is achieved relying on a weak notion of agency: following a biology inspiration, agents execute simple and pre-defined behaviour, out of which self-organisation is achieved by emergence [13]—ant foraging being a classical example. This approach however hardly applies to open and dynamic contexts in which what is the actual behaviour to be followed by a group of agents is to be decided (or even synthesised) at run-time.

On the other hand, a promising set of results towards addressing solid engineering of open self-organising systems are being achieved under the umbrella of *aggregate programming* [5]. Its main idea is to shift the focus of system programming from the individual's viewpoint to the aggregate viewpoint: one no

Q. Chen et al. (Eds.): PRIMA 2015, LNAI 9387, pp. 49–64, 2015.
DOI: 10.1007/978-3-319-25524-8_4

longer programs the single entity's computational and interactive behaviour, but rather programs the aggregate (i.e., the collective). This is achieved by abstracting away from the discrete nature of computational networks, by assuming that the overall executing "machine" is a sort of computational continuum able to manipulate distributed data structures: self-organisation mechanisms sit under the hood, and are the key for automatically turning aggregate specifications into individual behaviour. Aggregate programming is grounded in the computational field calculus [10], its incarnation in the Protelis programming language [29], on studies focussing on formal assessment of resiliency properties [35], and building blocks and libraries built on top to support applications in the context of large scale situated systems [3].

This paper aims at bridging the apparently significant gap between aggregate programming and agent autonomy, paving the way towards a fruitful cooperation by which stronger notions of agents (including deliberation and planning capabilities) can closely relate to self-organisation mechanisms. This is achieved by considering an aggregate program as a plan, what we call an "aggregate plan", operationally guiding the cooperative behaviour of a team of agents. Agents can create aggregate plans or receive them from peers, and can deliberate to execute them or not in different moments of time. The set of agents executing an aggregate plan forms a cooperating "dynamic team", coherently bringing about the social goal that the plan is meant to achieve, typically expressed in terms of a final distributed data structure used as input for other processes or to feed actuators (i.e., to make agents/devices move). The inner mechanisms of aggregate computing smoothly support entering/quitting the team, making overall behaviour spontaneously adapt to such dynamism as well as being resilient to changes in environment conditions.

The remainder of this paper is organised as follows: Section 2 overviews aggregate computing, Section 3 compares it with multi-agent systems and illustrates the aggregate plan idea, Section 4 described an example scenario of a distributed rescue activity by autonomous entities, Section 5 compares with alternative MAS approaches, and finally Section 5 concludes and discusses future works.

2 Aggregate Computing

Most paradigms of distributed systems development, there including the multi-agent system approach, are based on the idea of programming each single individual of the system, in terms of its computational behaviour (goals, plans, algorithm, interaction protocol), typically considering a finite number of "roles", i.e., individual classes. This approach is argued to be problematic: it makes it complicated to reason in terms of the effect of composing behaviours, and it forces the programmer to mix different concerns of resiliency and coordination—using middlewares that externalise coordination abstractions and interaction mechanisms only partially alleviate the problem [38,7].

These limits are widely recognised, and motivated work toward aggregate programming across a variety of different domains, as surveyed in [2]. Historically

such works addressed different facets of the problem: making device interaction implicit (e.g., TOTA [23]), providing means to compose geometric and topological constructions (e.g., Origami Shape Language [25]), providing means for summarising from space-time regions of the environment and streaming these summaries to other regions (e.g., TinyDB [21]), automatically splitting computational behaviour for cloud-style execution (e.g., MapReduce [11]), and providing generalisable constructs for space-time computing (e.g., Proto [24]).

Aggregate computing, based on the field calculus computational model [10] and its embodiment in Protelis programming language [29], lies in the above approaches and attempts a generalisation starting from the works on space-time computing [5], which are explicitly designed for distributed operation in a physical environment filled with embedded devices.

2.1 Computing at the Aggregate Level

The whole approach starts from the observation that the complexity of large-scale situated systems must be properly hidden "under-the-hood" of the programming model, so that composability of collective behaviour can be more easily supported and allow to better address the construction of complex systems. Aggregate programming is then based on the idea that the "machine" being programmed is a region of the computational environment whose specific details are abstracted away (perhaps even to a pure and uniform spatial continuum): the program is specified as a manipulation of data constructs with spatial and temporal extent across that region. Practically, since such "machine" is ultimately a collection of communicating devices, the semantics of aggregate programming is given as a mapping to a self-organising algorithm involving local interactions between such devices.

As an example, consider the problem of designing crowd steering services based on fully distributed, peer-to-peer interactions between crowd members' smart-phones. In this example, smart-phones could interact to collectively estimate the density and distribution of crowding, seen as a distributed data structure mapping each point of space to a real-value indicating the crowd estimation, namely, a *computational field* (or simply *field*) of reals [23,10]. This can be in turn used as input for actual steering services: warning systems for people nearby dense regions (producing a field of booleans holding true where warning has to be set), dispersal systems to avoid present or future congestion (producing a field of directions suggested to people via their smartphones), steering services to reach points-of-interest (POI) avoiding crowded areas (producing a field of pairs of direction and POI name). Building such services in a fully-distributed, resilient, and composable/reusable way is very difficult, as it comes to achieve self-* behaviour by careful design of each device's interaction with neighbours. With aggregate programming, on the other hand, one instead naturally reasons in terms of an incremental construction of continuous-like computational fields, with the programming platform taking care of turning aggregate programs into programs for the single device.

2.2 Constructs

The *field calculus* [10] captures the key ingredients of aggregate computation into a tiny language suitable for grounding programming and reasoning about correctness – recent works addressed type soundness [10] and self-stabilisation [35] – and is then incarnated into a Java-oriented language called Protelis [29], which we here use for explanation purposes. The unifying abstraction is that of computational field, and every computation (atomic or composite) is about functionally creating fields out of fields. Hence, a program is made of an expression `e` to be evaluated in space-time (ideally, in a continuum space-time, practically, in asynchronous rounds in each device of the network) and returning a field evolution. Four mechanisms are defined to hierarchically compose expressions out of values and variables, each providing a possible syntactic structure for `e`.

Application: $\lambda(\texttt{e}_1, \ldots, \texttt{e}_n)$ applies "functional value" λ to arguments $\texttt{e}_1, \ldots, \texttt{e}_n$, using call-by-value semantics and in a point-wise manner (output in a space-time point depend on inputs at the same point). λ can either be a "built-in" primitive (any non-aggregate operation to be executed locally, like mathematical, logical, or algorithmic functions, or calls to sensors and actuators), a user-defined function (that encapsulates reusable behaviour), or an anonymous function value $\texttt{(x}_1, \ldots, \texttt{x}_n\texttt{)->e}$ (treated as a value, and hence possibly passed also as argument, and ultimately, spread to neighbours to achieve open models of code deployment [10])—in the latter case Protelis ad-hoc syntax is $\lambda.\texttt{apply(e}_1, \ldots, \texttt{e}_n\texttt{)}$.

Dynamics: `rep(x<-v){e}` defines a local state variable `x` initialised with value `v` and updated at each node's computation round with the result of evaluating the update expression `e` (which mentions `x` to mean the old value).

Interaction: `nbr(e)` gathers by observation a map at each neighbour to its latest resulting value of evaluating `e`. A special set of built-in "hood" functions can then be used to summarise such maps back to ordinary expressions, e.g., `minHood(m)` finds the minimum value in the range of map `m`.

Restriction: $\texttt{if(e)\{e}_1\texttt{\} else \{e}_2\texttt{\}}$ implements branching by partitioning the network into two regions: where `e` evaluates to true $\texttt{e}_1$ is evaluated, elsewhere $\texttt{e}_2$ is evaluated. Notably, because `if` is implemented by partition, the expressions in the two branches are encapsulated and no action taken by them can have effects outside of the partition.

The above informal description roughly amounts to a denotational semantics of Protelis, given as a transformation of data structures dislocated in space-time [5]. An operational semantics, describing an equivalent system of local operations and message passing between devices [10], can be sketched as follows. Given a network of interconnected devices D that runs a main expression $\texttt{e}_0$, computation proceeds by asynchronous rounds in which a device $\delta \in D$ evaluates $\texttt{e}_0$. The output of each round at a device is an ordered tree of values, called *value-tree*, tracking the result of computing each sub-expression encountered during

evaluation of e_0. Such an evaluation is performed against the most recently received value-trees of neighbours, and the produced value-tree is conversely made available to all neighbours (e.g., via broadcast in compressed form) for their next round. Most specifically: `nbr(e)` uses the most recent value of `e` at the same position in its neighbours' value-trees, `rep(x<-v){e}` uses the value of `x` from the previous round, and `if(e){e1}else{e2}` completely erases the non-taken branch in the value-tree (allowing interactions through construct `nbr` with only neighbours that took the same branch, called "aligned neighbours").

2.3 Building Blocks and APIs

An example of aggregate program is the definition of a general building block `G` as reported in Figure 1(top) and thoroughly discussed in [4]. It is a highly reusable "spreading" operator executing two tasks: it computes a field of shortest-path distances from a `source` region (indicated as a boolean field holding true on sources) according to the supplied function `metric` (yielding a map from neighbours to a distance value), then propagates values along the gradient of the distance field away from source, beginning with value `initial` and accumulating along the gradient with function `accumulate`. A complementary operator is `C`, which accumulates information back to the `source` down the gradient of a supplied `potential` field; beginning with an idempotent `null`, at each device, the `local` value is combined with "uphill" values using a commutative and associative function `accumulate`, to produce a cumulative value at each device in the `source`. Another operator `S` can be used to elect a set of leaders with approximate distance `grain` from each other.

On top of such building blocks one can incrementally define general-purpose APIs. Some examples are shown in Figure 1(center), which culminate in functions `share` and `meanPathObstacles`. The former is used to gather information from an input field with a suitable accumulation function, broadcast it back so that all devices agree on the result, and do so by sub-regions of a given size partitioning the whole network. The latter computes in a fully-distributed and network-independent way the complex task of gathering in a source node the average amount of "obstacle" nodes (e.g. nodes sensing high traffic, or high-pollution) that one would encounter if travelling towards a destination according to a shortest-path: such information could be used to estimate the appropriateness of moving towards that destination.

Finally, to support openness and dynamism of code injection and management, our model support higher-order functions, allowing code (i.e., functions) to be passed around and be treated as data to diffuse. This allows to store a minimal code in each device, as show in Figure 1(bottom): function `deploy` is used to let function `g` be spread from sources and be executed remotely, while `virtual-machine` uses it to extract from environmental sensors the function to be injected, the injection point, and the range of diffusion. This means that a complex behaviour like that of functions `share` and `meanPathObstacles` needs not be statically present in each device, but could have been injected dynamically, received, and then executed by the set of involved devices.

```
// Spreads 'initial' out of 'source', using a given 'metric' and 'accumulate' update function
def G(source, initial, metric, accumulate) {
  rep(dv <- [Infinity, initial]) { // generates a field of pairs: distance + value
    mux(source) { // mux is a built-in ternary conditional operator
      [0, initial] // value of the field at a source
    } else { // lexicographic minimum of pairs obtained from neighbours
      minHood([nbr(dv.get(0)) + metric.apply(), accumulate.apply(nbr(dv.get(1)))])
    } // adding distance and accumulating.. then selcting min
  }.get(1) // of the pair only second component is returned
}
// Spreads 'initial' out of 'source', using a given 'metric' and 'accumulate' update function
def C(potential, accumulate, local, null) { ... }
// Elects leaders distant approximately 'grain' using a given 'metric'
def S(grain, metric) { ... }
```

```
// Computes minimum distance to 'source'
def distanceTo(source) {
  G(source, 0, () -> {nbrRange}, (v) -> {v + nbrRange})
}
// Broadcasts 'value' out of 'source'
def broadcast(source, value) {
  G(source, value, () -> { nbrRange }, (v) -> {v})
}
// Share values obtained 'accumulation' over 'field' done into regions of 'regionSize'
def share(field,accumulation,null,regionSize){
  let leaders = S(regionSize,() -> { nbrRange });
  broadcast(leaders,C(distanceTo(leaders),accumulate,field,null))
}
// Activates nodes on a path region with 'width' size connecting 'src' and 'dst'
def channel(src, dest, width) {
  distanceTo(src) + distanceTo(dest) <= broadcast(src,distanceTo(dest)) + width
}
// Gathers in sink the sum of 'value' across 'region'
def summarize(sink, region, value) {
  C(if (region) {distanceTo(sink)} else {Infinity}, +, value, 0)
}
// Gathers average level of obstacles 'obs' in the 'size'-path from 'src' to 'dest'
def meanPathObstacles(src, dest, size, obs) {
  summarize(src, channel(src, dest, size), obs)/summarize(src, channel(src, dest, size), 1)
}
```

```
// Evaluate a function field, running 'g' within 'range' meters from 'source', 'no-op' elsewhere
def deploy (range, source, g, no-op) {
    if (distance-to(source) < range) { broadcast(source,g).apply() } else {no-op.apply() }
}
// The entry-point function executed to run the virtual machine on each device
def virtual-machine () {
    deploy(sns-range, sns-injection-point, sns-injected-fun, ()->0)
}
```

Fig. 1. Building blocks `G`, `C`, and `S` (top), elements of upper-level APIs (center), VM bootstrapping code (bottom)

As one may note, an aggregate program never explicitly mentions a device's individual behaviour: it completely abstracts away from inner details of network shape and communication technology, and only assumes the network is dense enough for devices to cooperate by proximity-based interaction. Additionally, it is showed that operators `G`, `C` and `S`, along with construct if and function application mechanisms, form a self-stabilising set [4], hence any complex behaviour

built on top is self-stabilising to any change in the environment, guaranteeing resilience to faults and unpredicted transitory changes.

3 Aggregate Computing and Multi-agent Systems

In this section we draw a bridge between the agent-based approach and aggregate computing: after discussing commonalities and differences, we introduce and discuss the notion of aggregate plan.

3.1 Multi-agent Systems vs Aggregate Computing

Multi-agent systems and aggregate computing have some common assumptions that worth being recapped, and which form the basic prerequisite for identifying a common operating framework. First, they both aim at distributed solutions of complex problems, namely, by cooperation of individuals that, though being selfish, they are also social and hence be willing to bring about "social" goals and objectives. Second, they both assume agents are situated in a physical or logical environment, and work by perceiving a local context and acting on it, there including exchanging messages and sensing/acting on the environment. Finally, both approaches have been used to achieve self-organisation, typically by engineering nature-inspired solutions (mostly from biology for agents, and from physics for aggregate computing [39]).

On the other hand, multi-agent systems and aggregate computing have key differences, both conceptually and methodologically. First, with aggregate computing one programs the collective behaviour of individuals, whereas most agent approaches provide languages to program an agent behaviour, either as a reactive component exchanging messages adhering to given protocols, or as a proactive component with declarative goals, a deliberation cycle, and carrying on plans. Traditionally, weak forms of aggregation are considered in the MAS community as well, including the use of coordination mechanisms and tools (e.g. via artifacts [27] or protocols [18]), social/organisational norms [1], commitments [22], and so on. However, they either provide mere declarative constraints to agent interaction (i.e., they do not operationally describe the aggregate behaviour to carry on), or manage interactions between a typically small number of agents.

Second, agents feature autonomy as key asset. At least in the "stronger" notion of agency, agents do not follow pre-determined algorithms, but provide mechanisms to dynamically adapt their behaviour to the specific contingency: they have some even minimal ability to negatively reply to an external request, and to deviate from a previously agreed cooperative behaviour. On the other hand, in aggregate computing, individuals execute the same program in quite a rigid fashion: it is thanks to higher-order functions as developed in [10] that individuals can be given very simple initial programs which are unique system-wise, and later can execute different programs as a result of what function they receive and start executing. This mechanism is actually key for the adoption of aggregate computing mechanisms in multi-agent systems.

3.2 Aggregate Programs as Collective Plans

Though many ways of integrating aggregate computing and MAS exist (see a discussion in last section), in this paper we develop on the notion of "aggregate plan": a plan that an agent can either create or receive from peers, and can deliberate to execute or not in different moments of time, and which specifies an aggregate behaviour for a team of agents.

Life-cycle of Aggregate Plans. In our model, aggregate plans are expressed by anonymous functions of the kind `()->e`, where `e` is a field expression possibly calling API functions available as part of each agent's library—functions like those shown in Figure 1 (center). One such plan can be created in two different ways, by suitable functions (whose detail we abstract away): first, it can be a sensor (like `sns-injected-function` in Figure 1 (bottom)) to model the plan being generated by the external world (i.e. a system programmer) and dynamically deployed; second, it can model a local planner (e.g., a function `plan-creation`) that synthesises a suitable plan for the situation at hand. When the plan is created, it should then be shared with other agents, typically by a broadcasting pattern, like the one implemented by function `deploy`—though, the full power of field calculus can be used to rely on more sophisticated techniques for constraining the target area of broadcasting.

Agents are to be programmed with a `virtual-machine`-like code that makes it participate to this broadcast pattern, so as to receive all plans produced remotely in the form of a field of pairs of a description of the plan and its implementation by the anonymous function. Among the plans currently available, by the restriction operator `if` the agent can autonomously decide which one to actually execute, using as condition the result of a built-in deliberation function that has access to the plan's description.

Note that if/when an aggregate plan is in execution, it will make the agent cooperatively work with all the other agents that are equally executing the same aggregate plan. This "dynamic team" will then coherently bring about the social goal that this plan is meant to achieve, typically expressed in terms of a final distributed data structure, used as input for other processes or to feed actuators (i.e., to make agents/devices move). The inner mechanisms of aggregate computing smoothly support entering/quitting the team, making overall behaviour spontaneously self-organise to such dynamism.

Mapping Constructs, and Libraries. As a plan is in execution, the operations of aggregate programming that it includes can be naturally understood as "instructions" for the single agent, as follows:

- Function application amounts to any pure computation an agent has to execute, there including algorithmic, deliberation, scheduling and planning activities, as well as local action and perception.
- Repetition construct is instead used to make some local result of execution of the aggregate plan persist over time, e.g. modelling belief update.

- Restriction can be used inside a plan to temporarily structure the plan in sub-plans, allowing each agent to decide which of them should be executed, i.e., which sub-team has to be dynamically joined.
- Neighbour field construction is the mechanism by which information about neighbour agents executing the same (sub-)plan can be observed, supporting the cooperation needed to make the plan be considered as an aggregate one.

As explained in previous section, one of the assets of aggregate programming is its ability of defining libraries of reusable components of collective behaviour, with formally provable resilience properties. Seen in the context of agent programming, such libraries can be used as libraries of reusable aggregate plans, built on top of building blocks:

- Building block `G` is at the basis of libraries of "distributed action", namely, cooperative behaviour aimed at acting over the environment or sets of agents in a distributed way. Examples include, other than broadcasting information (`broadcast`) and reifying distances (`distanceTo`), also the possibility of forecasting events, creating network partitions, clusters or topological regions in general.
- Building block `C` conversely supports libraries of "distributed perception", namely, cooperative behaviour aimed at perceiving the environment or information about a set of agents in a distributed way. They allow gathering "aggregate" values over space-time regions, like sums, average, maximum, semantic combination [32], as well as computing consensus values [12].
- Building block `S` supports libraries requiring forms of explicit or implicit leader elections, often needed to aggregate and then spread-back consensus values.

The combination of building blocks `G`, `C`, `S`, and others [4], allow to define more complex elements of collective adaptive behaviour, generally used to intercept distributed events and situations, compute/plan response actions, and actuate them collectively.

4 Case Study

As an exemplification of the above mentioned concepts, we propose a cooperative teamwork case study. We imagine that a situation of emergency occurs in a urban scenario. A group of rescuers is responsible of visiting the areas where an unknown number of injured victims are likely to be located, and decide in which order to assist them (e.g., to complete a *triage*). We suppose rescuers carry a smart device, and are able to communicate with each other the position of victims in their field of view. Such devices are equipped with a small VM code (a minimal aggreate plan), which is responsible of computing and selecting a collaborative strategy for exploring the area, and of displaying a coordinate to go to. Rescuers do not initially know the exact position of victims: they are required to get to the area by visiting a set of coordinates assigned by the mission control, explore the surroundings, and assist any people they encounter that require treatment.

4.1 An Aggregate Computing Approach

Our devices start with a simple plan, suggesting directions to reach given coordinates assigned by the mission control.

As a rescuer sees a victim, it creates and injects a second, more advanced plan, working as follows: if the rescuer sees a victim and nobody else is assisting her, then it simply takes charge of assisting. If somebody else is already assisting her, instead, the rescuer moves towards the closest known victim if any, relying on the aggregate knowledge of all the other rescuers, namely taking into account all the victims that had been seen by any other rescuer—namely, performing a distributed observation. As a consequence, rescuers will tend to come close to the areas where victims have already been found but not assisted yet. The idea behind the plan is that it is likely that many victims are grouped, and as such calling for more rescuers may raise the probability of finding them all. If no victims have been discovered or all have been assisted, then the dynamic aggregate plan is quit, and the initial plan of reaching the target destination (former exploration strategy) is executed.

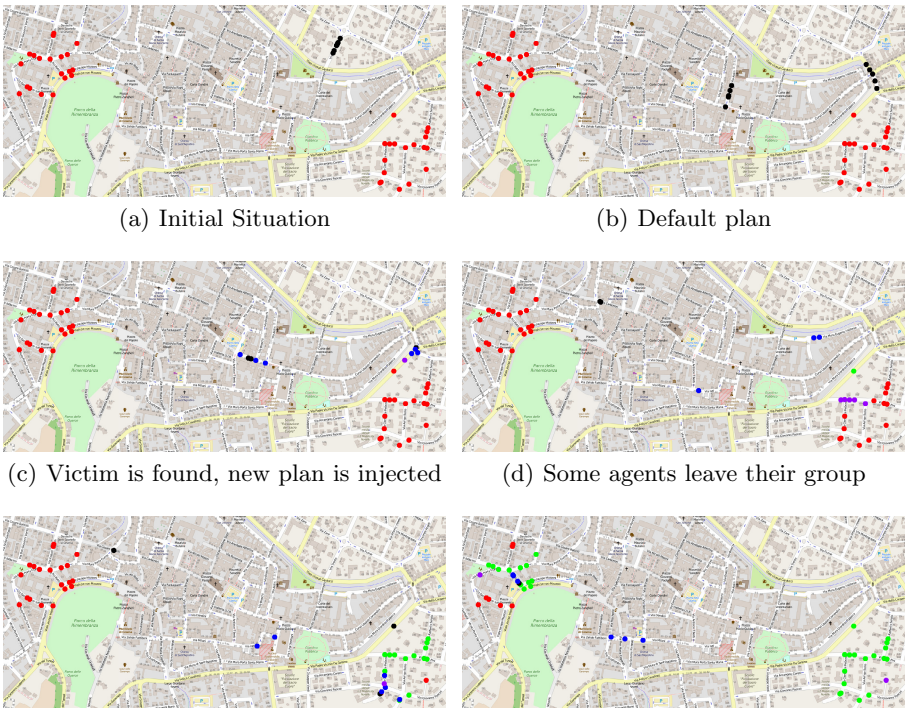
The collaborative strategy requires agents to collectively compute fields like `remaining`, mapping each agent to the list of positions of nearby victims to be assisted. Building, maintaining and manipulating distributed and situated data structures like this one, and do so in a resilient way [35], is a fundamental brick of any aggregate plan, and specifically, of the dynamic plan that in this scenario an agent may decide to play. Aggregate computing can be used to smoothly solve the problem by the following sub-plan:

- by function `share` a field of known `victims` can be created by the union of single agent's knowledge about victims, as reflected by their visual perception;
- similarly, again by function `share`, a field of victims currently `assisted` can be created based on information coming from the agents actually assisting;
- by set subtraction, the field `remaining` is built that provides information about victims not assisted yet;
- if field `victims` is empty, the plan is quit;
- otherwise, if there are no known `remaining`, the agent moves to the closest `assisted` – where it is likely to find new victims;
- otherwise, the agent moves to the closest `remaining`, assisting the victim as she is reached.

Once this plan and the original one are alive, each agent can autonomously decide which one to follow.

4.2 Simulation

We have chosen to simulate our case study in an urban environment. There, two groups of victims are displaced, and a group of rescuers starts its mission from in-between those two groups. Initially, coordinates are generated by the mission control, and are given to rescuers. Once the first victim is encountered,



(a) Initial Situation (b) Default plan

(c) Victim is found, new plan is injected (d) Some agents leave their group

(e) Assistance is performed collaboratively (f) Victims are found on a new area, and some agents move towards them

Fig. 2. Simulation of the case study. Rescuers that follow their initial plan are pictured in black, rescuers that are acting collaboratively are in blue, and rescuers that are assisting are marked in purple. Victims are in red, until they receive assistance and switch to green. The rescuers initially split in two groups, but when the first victim is found, some of those going towards the other victim group decide to change their plan and act collaboratively.

the aggregate plan becomes available. To simulate autonomous behaviour, we assign a certain probability p that an agent accepts to follow the aggregate plan: if it does not, it keeps moving towards the given coordinates. Each ten minutes each agent reconsiders its decision to follow the aggregate plan, again according to probability p.

We used Alchemist [28] as simulation platform. Alchemist supports simulating on real-world maps, exploiting OpenStreetMaps data and imagery, and leveraging Graphhopper for navigating agents along streets. The actual city used as scenario is the Italian city of Cesena.

Section 4.2 shows some snapshots from a run executed with $p = 0.75$. Rescuers (black dots) visit the coordinates starting from a random one; as such, they initially split in two sub-groups. At some point, one of them finds a victim. The second plan becomes available, and some of the agents, both in the close

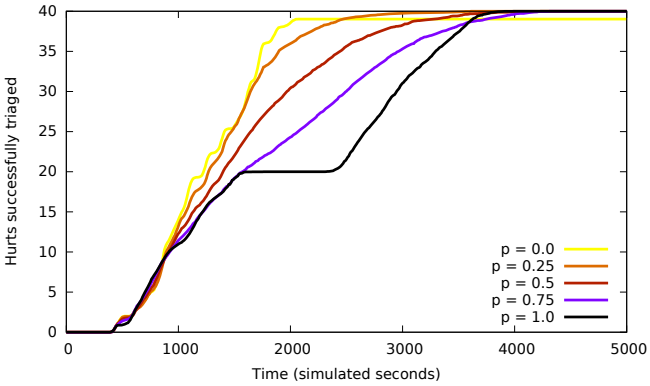


Fig. 3. Impact of our measure of autonomy p on the numer of victims assisted with time. Mean of 100 runs for each value of p.

sub-group and in the group far away change their behaviour and start walking towards such a victim. The number of rescuers that change their original direction and follows the new plan raises with p. With $p = 1$, all of them start chasing a victim as soon as found, with $p = 0$ none of them does, and every one continues to visit its waypoints. $1 - p$ can be seen as a measure of the likelihood that the suggested plan is rejected by the freedom of autonomy: in this sense, it reflects a measure of the level of autonomy of each agent.

Figure 3 shows how p impacts the performance of the system in our scenario. Our expectation is that the results with very low p strongly depend on how fortunate is the initial waypoints choice. In case the selection is rough or approximate, the rescuers may end up not finding all the victims. The chart confirms this expectation: the line for $p = 0$ rises very quickly (agents divide their load pretty evenly), but the final result is that in 99 out of 100 runs some of the victims are not discovered. It is worth observing the behaviour for $p = 1$: initially, all the rescuers walk towards the first group that is found. Only when this group is completely assisted, they start looking for more victims on different places, hence the flat part of the black line. Once the second group is found, all the rescuers are attracted towards that area, and the assisting procedure is quickly completed: as shown, in fact, the number of successfully assisted victims steeply rises. Intermediate values of p show a progressive change in behaviour between the two extremes. A very solid and globally quick result is obtained with $p = 0.25$, suggesting that the system takes advantage from both autonomy in choice and collective decisions.

5 Related MAS Approaches

Alternative ways to implement the case study and similar systems with MASs range from purely subjective approaches (agents realise coordination) to

objective approaches (the coordination burden is externalised to coordination media and/or organisation mechanisms) [26].

The simplest example for the subjective case is given by a well-known platform like JADE [6], which does not provide any specific support to coordination but only relies on speech-act/FIPA based communication protocols. In this case, the design of such protocols should take into the account all the issues that we implicitly manage at the aggregate level and achieve self-organisation. So, our approach sits at a rather higher abstraction level and hence defines a much more convenient engineering framework—JADE could of course be possibly used as a low-level platform to support our agent-to-agent interactions.

More complex subjective approaches exploit Distributed AI techniques for MAS coordination and teamwork—such as distributed scheduling and planning [16,20,33]. A main example for the first case is given by Generalized Partial Global Planning (GPGP) and the TAEMS Framework for Distributed Multi-agent systems [19]. A main example for the second case is given by approaches based on Distributed (Partially Observable) Markov Decision Problems (Distributed (PO)MDPs) and Distributed Constraint Optimization Problems (DCOPs) [34]. While these approaches have been proven to be effective in realistic domains, taking into the account the uncertainty of agent's actions and observations, they typically fail to scale up to large numbers of agents [34], which aggregate computing smoothly addresses by construction.

On the objective side, a viable approach would be to rely on specific coordination artifacts [27] (like tuple spaces and their variants). Using models/infrastructures like TuCSoN [14,36], or the chemical-inspired SAPERE approach [37], one could couple each device with a networked shared space in which tuple-like data chunks can be injected, observed, and get diffused, aggregated, and evaporated by local coordination rules. Although proper coordination rules could in principle be instrumented to achieve a similar management of computational fields, the functional nature of aggregate computing – that is crucial to support reusability and composability – is difficult to mimick, if not by an automatic compilation process.

A different objective approach is rooted on organisation/normative models and structures, e.g., using Moise and a supporting platform like JaCaMo [7]—which integrates the organisation dimension (based on Moise) with the agent dimension (based on the Jason agent programming language) and the environment one (based on the CArtAgO framework). In this case, the solution would account to explicitly define the coordination among agents in terms of the structural/functional/normative view of the organisation of the MAS. The coordination aspect would be handled in particular by defining missions and social schemas. The level of self-organisation supported in this case is limited to the way in which individual agents are capable to achieve the individual goals that are assigned and distributed by the organisational infrastructure, executing the "organisation program". The functional description of the coordination to be enacted at the organisation level – specified by missions – is essentially a shared plan explicitly defining which goals must be done and in which order.

This approach is not feasible in the application domains for which aggregate computing has been devised, where it is not possible (or feasible) in general to define a priori such a shared static global plan. That is, it is possible to specify what is the desired goal (e.g. what kind of spatial distribution of the human/agents we want to obtain), but not its functional decomposition in subtasks/subgoals to be assigned to the individual agents/roles.

6 Conclusion and Future Works

In this work we started analysing a bridge between multi-agent systems to aggregate programming. Aggregate programming can be useful in MAS as an approach that allows to specify the behaviour and goal of the MAS at the global level, as a kind of *shared plan* abstracting from the individual autonomous behaviour. This makes our work strongly related to existing literature in MAS dealing with cooperative planning in partially observable stochastic domains [17], and decision making in collaborative contexts adopting formalized frameworks based on a notion of shared plans [15]. In that perspective, there are many interesting directions that can be explored in future work. A first one is to explore aggregates composed by intelligent agents based on specific model/architecture such as the BDI one, promoting a notion of *aggregate stance* that may integrate with classical *intentional stance* [8]. A second direction is to explore the link with approaches that investigate the use *stigmergy* and coordination based on implicit communication in the context of aggregates of intelligent agents [9,30]; Finally, it would be interesting to integrate aggregate programming with agent-oriented programming [31] and current agent programming languages, making it possible to conceive agent programs that e.g. exploit both local plans – like in the case of a BDI agent language like Jason – and global plans as conceived by the aggregate level.

References

1. Artikis, A., Sergot, M.J., Pitt, J.V.: Specifying norm-governed computational societies. ACM Trans. Comput. Log. 10(1) (2009)
2. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013). http://arxiv.org/abs/1202.5509
3. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. IEEE Computer (2015)
4. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: IEEE Workshop on Foundations of Complex Adaptive Systems (FOCAS) (2014)
5. Beal, J., Viroli, M.: Space–time programming. Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 373(2046) (2015)

6. Bellifemine, F.L., Poggi, A., Rimassa, G.: Developing multi-agent systems with JADE. In: Castelfranchi, C., Lespérance, Y. (eds.) ATAL 2000. LNCS (LNAI), vol. 1986, p. 89. Springer, Heidelberg (2001)
7. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with jacamo. Science of Computer Programming **78**(6), 747–761 (2013)
8. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press, Nov. 1987
9. Castelfranchi, C., Pezzulo, G., Tummolini, L.: Behavioral implicit communication (BIC): Communicating with smart environments via our practical behavior and its traces. International Journal of Ambient Computing and Intelligence **2**(1), 1–12 (2010)
10. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: a higher-order calculus of computational fields. In: Graf, S., Viswanathan, M. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. LNCS, vol. 9039, pp. 113–128. Springer, Heidelberg (2015)
11. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
12. Elhage, N., Beal, J.: Laplacian-based consensus on spatial computers. In: 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), pp. 907–914. IFAAMAS (2010)
13. Fernandez-Marquez, J.L., Serugendo, G.D.M., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Natural Computing **12**(1), 43–67 (2013)
14. Gardelli, L., Viroli, M., Casadei, M., Omicini, A.: Designing self-organising environments with agents and artefacts: A simulation-driven approach. International Journal of Agent-Oriented Software Engineering **2**(2), 171–195 (2008)
15. Grosz, B.J., Hunsberger, L., Kraus, S.: Planning and acting together. AI Magazine **20**(4), 23–34 (1999)
16. Grosz, B.J., Kraus, S.: Collaborative plans for complex group action. Artificial Intelligence **86**(2), 269–357 (1996)
17. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1–2), 99–134 (1998)
18. Kalia, A.K., Singh, M.P.: Muon: designing multiagent communication protocols from interaction scenarios. Autonomous Agents and Multi-Agent Systems **29**(4), 621–657 (2015)
19. Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., Prasad, M., Raja, A., Vincent, R., Xuan, P., Zhang, X.: Evolution of the GPGP/TAEMS domain-independent coordination framework. Autonomous Agents and Multi-Agent Systems **9**(1–2), 87–143 (2004)
20. Levesque, H.J., Cohen, P.R., Nunes, J.H.T.: On acting together. In: Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI 1990, vol. 1, pp. 94–99. AAAI Press (1990)
21. Madden, S.R., Szewczyk, R., Franklin, M.J., Culler, D.: Supporting aggregate queries over ad-hoc wireless sensor networks. In: Workshop on Mobile Computing and Systems Applications (2002)
22. Mallyak, A.U., Singh, M.P.: An algebra for commitment protocols. Autonomous Agents and Multi-Agent Systems **14**(2), 143–163 (2007)
23. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. on Software Engineering Methodologies **18**(4), 1–56 (2009)

24. MIT Proto. http://proto.bbn.com/ (retrieved on January 1, 2012)
25. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. Ph.D. thesis, MIT (2001)
26. Omicini, A., Ossowski, S.: Objective versus subjective coordination in the engineering of agent systems. In: Klusch, M., Bergamaschi, S., Edwards, P., Petta, P. (eds.) Intelligent Information Agents. LNCS (LNAI), vol. 2586, pp. 179–202. Springer, Heidelberg (2003)
27. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems 17(3), June 2008
28. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with Alchemist. Journal of Simulation (2013)
29. Pianini, D., Viroli, M., Beal, J.: Protelis: Practical aggregate programming. In: Proceedings of ACM SAC 2015, pp. 1846–1853. ACM, Salamanca, Spain, (2015)
30. Ricci, A., Omicini, A., Viroli, M., Gardelli, L., Oliva, E.: Cognitive stigmergy: towards a framework based on agents and artifacts. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2006. LNCS (LNAI), vol. 4389, pp. 124–140. Springer, Heidelberg (2007)
31. Shoham, Y.: Agent-oriented programming. Artif. Intell. **60**(1), 51–92 (1993)
32. Stevenson, G., Ye, J., Dobson, S., Pianini, D., Montagna, S., Viroli, M.: Combining self-organisation, context-awareness and semantic reasoning: the case of resource discovery in opportunistic networks. In: ACM SAC, pp. 1369–1376. ACM (2013)
33. Tambe, M.: Towards flexible teamwork. J. Artif. Int. Res. **7**(1), 83–124 (1997)
34. Taylor, M.E., Jain, M., Kiekintveld, C., Kwak, J., Yang, R., Yin, Z., Tambe, M.: Two decades of multiagent teamwork research: past, present, and future. In: Guttmann, C., Dignum, F., Georgeff, M. (eds.) CARE 2009 / 2010. LNCS, vol. 6066, pp. 137–151. Springer, Heidelberg (2011)
35. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organising systems by self-stabilising fields. In: IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2015) (2015)
36. Viroli, M., Casadei, M., Omicini, A.: A framework for modelling and implementing self-organising coordination. In: Proceedings of ACM SAC 2009, volume III, pp. 1353–1360. ACM, March 8–2, 2009
37. Viroli, M., Pianini, D., Montagna, S., Stevenson, G., Zambonelli, F.: A coordination model of pervasive service ecosystems. Science of Computer Programming, June 18, 2015
38. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Autonomous Agents and Multi-Agent Systems **14**(1), 5–30 (2007)
39. Zambonelli, F., Viroli, M.: A survey on nature-inspired metaphors for pervasive service ecosystems. International Journal of Pervasive Computing and Communications **7**(3), 186–204 (2011)