

Combining Self-Organisation and Autonomic Computing in CASs with Aggregate-MAPE

Mirko Viroli^{*}, Antonio Bucchiarone[†], Danilo Pianini^{*}, Jacob Beal[‡]

^{*}Università di Bologna, Italy; Email: {mirko.viroli,danilo.pianini}@unibo.it

[†]Fondazione Bruno Kessler, Trento, Italy Email: bucciarone@fbk.eu

[‡]Raytheon BBN Technologies, USA; Email: jakebeal@bbn.com

Abstract—Aggregate computing is a recently proposed framework to build CASs (collective adaptive systems) by focussing on direct programming of ensembles so as to abstract away from individual devices and their single interaction acts: this approach is shown to streamline the identification of highly reusable block components, and support reasoning about their resiliency properties. Following this paradigm, in this paper we present a framework for bridging the gap between the MAPE (Monitor-Analyse-Plan-Execute) loop of autonomic computing managers, and fully-distributed self-organising CASs. This is achieved by seeing the collection of M components of each agent as an aggregate, amenable to a direct specification as overall CAS Monitoring behaviour, and similarly for A, P and E. As a result, a self-organising CAS can be programmed by clearly separating the M, A, P, and E parts of it; though each is expressed in terms of a collective behaviour. The proposed approach is exemplified with an application scenario of crowd dispersal in a large-scale smart-mobility application.

I. INTRODUCTION

Contemporary and future software systems are composed of large-scale ensembles [1] of widely distributed, largely autonomous and heterogeneous agents situated in both the physical world and in back-end computer systems. Those ensembles are often open-boundary and multi-owned, resulting in the lack of a viable central point of command and control. Moreover, human interaction via ubiquitous computing devices is often deeply embedded and must be considered as an integral part of these kinds of system. We are therefore increasingly looking at, and designing, collective and adaptive socio-technical applications built on top of large-scale decentralised distributed computing systems [2]. These systems can be effectively managed only via decentralised adaptation. Such adaptation must be itself collective, that is, multiple agents must adapt simultaneously in a way that, on the one hand, properly addresses critical runtime conditions, while, on the other hand, does not break the working consistency of the ensemble, preserving the collaboration and its benefits.

There have been many theoretical and methodological solutions in the field of collective adaptation which can deal with large-scale distributed and heterogeneous software systems. However, most of the proposed solutions work under an architectural model in which the adaptation knowledge is logically centralised, and the control of adaptation is exerted centrally. There is still a lack of understanding on how to

engineer Collective Adaptive Systems (CAS), in which a central control is not possible.

Aggregate computing is a recently proposed framework to build CASs (collective adaptive systems) based on the idea of directly engineering an ensemble by abstracting away from the individuals and their single interaction acts [3], [4]. From the foundational viewpoint, aggregate computing is about defining computations in terms of functions manipulating global-level, distributed data structures (i.e., computational fields [5], [6]); most specifically, each piece of collective behaviour is seen as a declaratively-specified function from fields to fields, equipped with a “compilation technique” [7] that turns it into single operations (sensing, computing, sending messages) to be executed by each agent. This approach is shown to streamline the identification of highly reusable block components, for which resiliency properties can be formally proved [3].

As a contribution towards better engineering of CASs, and following the aggregate computing paradigm, we here address the problem of fully distributing the typical adaptation technique of autonomic computing, that of MAPE (Monitor-Analyse-Plan-Execute) control loop [8]. This is achieved by an aggregate-MAPE framework, in which one sees the collection of M components of each agent as a single aggregate, amenable to a direct specification of overall CAS Monitoring behaviour, and similarly for A, P and E. As a result, a self-organising CAS can be programmed by clearly separating the M, A, P, and E parts of it, and expressing each of them in terms of a collective behaviour, to be carried on by the whole set of agents (or a dynamically selected subset) via implicitly occurring message exchanges.

The remainder of the paper is organised as follows: Section 2 provides background on CASs, MAPE, and previous attempts at distributing control loops, Section 3 discusses essential elements of aggregate computing, Section 4 introduces our proposal of aggregate-MAPE, Section 5 discusses the case study, and finally Section 6 concludes with final remarks.

II. BACKGROUND

A. Collective Adaptation in Multi-Agent Systems

The term *ensemble* has recently been introduced in the literature to denote very large-scale systems of systems

that may present substantial socio-technical embedding [1]. They typify systems with complex design, engineering and management, whose level of complexity comes specifically from bringing together and combining in the same operating environment many heterogeneous and autonomous components, systems, and users, with their specific concerns. To be robust against the high degree of unpredictability and dynamism of their operating environments, and to sustain the continuous variations induced by their socio-technical nature, ensembles need to *self-adapt*.

Self-adaptation is an important feature of many complex software systems, but it is often seen as a means to automate management activities in order to meet desired requirements, such as minimising resource usage and costs (e.g. [9]). In an ensemble-based approach, self-adaptation is instead a feature of the collectiveness. Individual agents may “opportunistically” enter in an ensemble and self-adapt in order to leverage other agents’ resources, functionalities and capabilities to perform their task more efficiently or effectively. However, the collaborative nature of the ensemble makes self-adaptation much trickier. Changes in the behaviour of one agent, as a result of its own self-adaptation, may break the consistency of the whole collaboration, or have negative repercussions on other agents in the ensemble. Adaptation must therefore be *collective*. Agents within an ensemble must be able to self-adapt simultaneously but, at the same time, preserve the collaboration and benefits of the ensemble they are within. Self-adaptation of an individual agent is therefore not only limited to the achievement of its own respective goals but also to the fulfilment of emerging goals of the dynamically formed ensembles (i.e., *self-organisation*).

Existing works for distributed self-adaptive systems are typically based on multi-agent and MAPE loop paradigms [10], [11]. More specifically, the system is decomposed in self-handling software units, which collaborate and coordinate in a distributed way. In [12], for example, the life-cycle of each agent is decomposed into three steps: “Perceive - Decide - Act” where the “Decide” phase is the key step where the agent chooses which action it has to perform using its partial perceptions.

Several research effort has been devoted to multi-agent *coordination* problems. Decentralisation of control implies a style of coordination in which the agents cooperate as peers with respect to each other, and no agent has global control over the system or global knowledge about the system. As a result, complex interactions are necessary to achieve consensus since there is no single agent that can make a centralised decision. In the case of mobile applications, agents have to take into account the distribution of the nodes in physical space and other properties of the environment, which add extra layers of complexity [11].

“Since development of distributed multi-agent systems is difficult, usually middleware is used to support the application developer” [11]. Coordination solutions provided for distributed computing support could be also exploited

(e.g., [13]). The considered applications basically consist of distinct application components that cooperate as peers to reach the overall goal of the application. “No single component has global control over or knowledge about the system. Decentralization of control typically increases both the importance and the complexity of coordination in the application” [11].

In [14], a rigorous approach is defined to decentralise the control loops of distributed self-adaptive software used in mission-critical applications. Specifically, it uses quantitative verification at runtime, first to agree individual component contributions to meeting system level quality-of-service requirements, and then to ensure that components achieve their agreed contributions in the presence of changes and failures. *All verification operations are carried out locally, using component-level models, and communication between components is infrequent.*

The work in [15] provides a contribution towards the design and implementation of decentralised solutions for the autonomic management of large and highly dynamic distributed pervasive systems. Specifically, a fully decentralised middleware (GOPRIME) is proposed for the adaptive self-assembly of distributed services. Abstracting from characteristics of specific application domains, GOPRIME aims at managing distributed systems where a set of peers cooperatively work to accomplish specific tasks. In general, each peer possesses the knowledge about how to perform some tasks (offered services), but could require services offered by other peers to carry them out. In this context, the GOPRIME goal is to drive a self-assembly procedure among the peers, aimed at matching required and provided services. Moreover, it is assumed that the system operates under non-functional requirements concerning the quality of the offered services (QoS) (e.g., performance, dependability, cost) and/or the structure of the resulting assembly.

B. Self-Organized Ensembles

As we discussed in the previous section, existing approaches typically deal with multi-agent adaptive systems through isolated adaptation: each agent adapts itself independently from each other. However, we want to consider a more involved problem where even though the agents are generally autonomous, they dynamically form ensembles to gain otherwise impossible benefits.

Adherence to these collective rules temporarily reduces the flexibility of collaborating agents and has tremendous impact on how the agents adapt to dynamic changes. Isolated adaptation is no longer effective, and new approaches are needed: 1) *multiple* agents must adapt altogether and transactionally; and 2) some kind of *negotiation* must take place to decide on the changes to be applied on each side. Moreover, the need to have *collective adaptation* raises an important issue, VIZ identifying which parts of the system should be engaged in adaptation. This is an intricate task, since solving a problem may require to take actions at

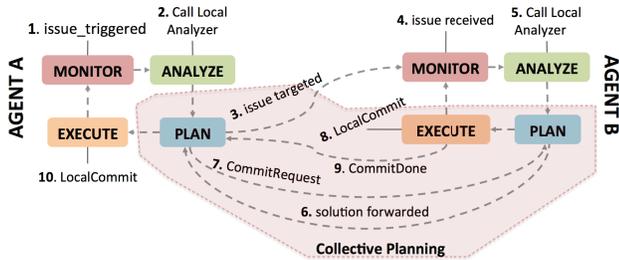


Fig. 1. Collective Planning according to the architecture proposed in [2].

different scales, involving several spontaneously created ensembles evolving over time.

In [2] an example model of collective adaptation is proposed that is built around the concept of *ensemble*, a collection of autonomous agents which collaborate to perform certain tasks. Each agent implements a MAPE loop that allows for the dynamic interaction with other agents. Figure 1 depicts the communication between two agents via their MAPE loops. During its normal execution, each agent is in the `Monitoring` state, while executing its tasks and monitoring the environment through active handlers. Issues can come both from the agent itself (`issue triggered` ①) or from a different entity, which asks support in order to solve an issue (`issue received` ④). At this point, the `Analyse` state is activated. On the basis of the triggered issue, the corresponding analyser is called (`Local Analyser` ② and ⑤). Here we enter the `Planning` state where all the agents involved in the issue resolution process (i.e., Agents A and B) will collaborate to solve the issue. If the solution provided by the `Local Analyser` foresees the involvement of other agents, the identified issues are triggered (`issues targeted` ③) to the involved agents in the resolution process (`issues received` ④). Once an agent receives feedback from the triggered agents (`solution forwarded` ⑥), it selects the better solution, using some form of multi-criteria ranking of alternatives (i.e., AHP [2]), and asks to the involved targets to commit their local solution (`Commit Request` ⑦); then it waits for their commit to be done (`Commit Done` ⑨), and eventually it commits its local solution (`Local Commit` ⑩). While waiting for a commit, the entity can receive a positive or a negative reply for its proposed solution. Whichever the case, it executes a solution commit, which will result empty in the negative case.

It is the goal of the present paper to fully develop on this idea, seeking for techniques to collectively execute the entire MAPE loop.

III. AGGREGATE COMPUTING

Most paradigms of distributed systems development, including the multi-agent system (MAS) approach, are grounded on the idea of programming each individual in the system with goals, plans, algorithm, and interaction protocol. However, individual-centric approaches of this sort are known to be problematic when it comes to reasoning

about the behavior of large-scale compositions of agents [6]. Building and generalising on previous approaches to organise collections of computational elements, *aggregate computing* is a framework for designing complex, large-scale ensembles of situated agents in a way that is effectively independent of (and as such, adapting to) the number and distribution of such agents [3]. It is based on the idea that the “abstract machine” one wants to program is not the single agent device, but rather the logically-single computational entity formed by the entire set of agents constituting a given team: an aggregate specification, hence, never mentions an individual’s actions or tasks, but always collective ones. Such computations express manipulations of physically-distributed data structures, called *computational fields* [5] (time-varying maps from devices to computational values—sensor data, temporaneous data values, knowledge), which are amenable to interpretation by the single agent as a local program, automatically dictating local computations and interactions with other neighbouring agents.

The field calculus (formalised in [7], and implemented as the Protelis language in [16] and the SCAFI framework in [?]) is the foundation of aggregate computing, as it provides the key mechanisms to work with computational fields, and on top of which resiliency properties can be proven by construction or formal reasoning (see e.g., [4], [17]). The core idea of field calculus is to express computations by a functional language with the “everything is a field” philosophy: reusable behaviour can be expressed in terms of declaratively-specified transformation from fields to fields; in fact, any field computation takes field as input and produces a field as output. For example, given an input of a Boolean field mapping certain devices of interest to `true` (representing any predicate over sensed values), an output field of estimated distances to the nearest such device can be constructed by iterative aggregation and spreading of messages between agents, in such a way that, as the input changes, the output changes to match the new situation. Such distances could then get used, for instance, to feed navigation services. This function can be programmed as follows:

```
def distanceTo(source){
  rep(d <- Infinity){
    mux(source) {0} else {minHood(nbrRange() + nbr(d))}
  }
}
```

This code estimates distance `d` to devices where `source` is `true`: it is initially infinity everywhere, and is computed over time using built-in functional selector `mux` to set sources to 0 and other devices by the triangle inequality, taking the minimum value obtained by adding the distance to each neighbour (as given by sensor `nbrRange()`) to its estimate of `d` (obtained by `nbr`).

IV. AGGREGATE MAPE

Following the direction of previous works discussed in Section II (such as [2]) we aim here at devising a fully

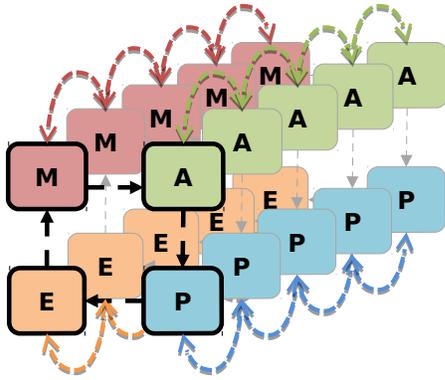


Fig. 2. An Aggregate MAPE is essentially made by n agents hosting 4 aggregate processes (for M, A, P and E): interactions between these 4 are explicitly programmed, while interactions across agents are implicitly defined by the aggregate computing model.

distributed version of a MAPE-loop, with the goal of providing fully distributed and resilient adaptation strategies for large-scale ensembles. An initial attempt with aggregate computing is described in [18], in which the planning stage of an agent deliberation loop is structured as an aggregate process producing the set of actions each agent has to execute across space and time.

Generalising this approach, we propose “aggregate-MAPE” as a framework in which all four M, A, P, and E components (referred to as the MAPE components in the following) are actually aggregate processes. Each of them defines a functionality at the global level (with computational fields as inputs and outputs): by aggregate computing, this means that the M components of all agents form an aggregate system bringing about the overall, distributed monitoring goal of the system, and where the information they should accordingly share is implicitly defined “under-the-hood” of the computational model; and similarly for A, P and E—see Figure 2. This approach calls for a design methodology in which the overall system behaviour is clearly divided from the beginning in the MAPE components, and where each has a well defined interface and can be engineered and tested in isolation, possibly relying on the library of reusable components available for aggregate computing [3]. Note that the whole issue of whether the MAPE components must be known before the application starts, or can be designed and injected on-the-fly, is completely orthogonal: as discussed in [18], [7], the aggregate computing toolchain allows one to make new code be injected, diffused, and executed on-the-fly, even only by the subset of agents deliberately deciding to be part of the sub-team bringing about a common goal.

Most specifically, design of an aggregate MAPE encompasses:

M Monitor is a process fed with local sensor values, and providing distributed sensing, in the form of one or multiple fields reifying the result of collective sensing; typically it includes mechanisms such as counting number of events, averaging a property, locally de-

```
def aggregateMAPE(m, a, p, e) {
  let monitoring = m.apply();
  let analysis = a.apply(monitoring);
  let plans = p.apply(analysis); // Returns a set of labels
  // Converts labels to actions
  let strategies = plans.map(self, label -> {
    e.apply(analysis, label)
  });
  // actual execution
  strategies.map(self, action -> {
    action -> {action.apply()}
  })
}
```

Fig. 3. Protelis code implementing the MAPE state machine. Protelis has a syntax reminiscent of Java 8: the arrow symbols (\rightarrow) prefixed with arguments and followed by a Protelis fragment enclosed in curly brackets declares an anonymous function.

tecting contingencies, etc.

- A** Analyse is a process taking the results of monitoring, digesting them according to the application needs, and providing a field of “alert” values spotting the relevant events that should trigger some adaptation; typically it includes mechanisms such as information fusion, complex correlation of spatio-temporal information, identification of patterns, etc.
- P** Plan is a process taking the results of analysis, crossing it with available adaptation resources, and providing a set of plans to subgroups of agents, in terms of a field of sets of “plan names”; it typically includes mechanisms such as network partitioning, distributed consensus, local deliberation of plans, etc.
- E** Execute is a process fed with sets of plan names, and able to interpret each as an aggregate process to be carried on by the associated subgroup of agents; typically it includes all actuation mechanisms for the application at hand, such as moving agents, providing suggestions in a person’s smartphone, controlling actuators deployed in the environment, etc.

The code in Figure 3 shows how a simple virtual-machine for aggregate MAPE can be written in Protelis [16], in terms of a function accepting four arguments and executing them in turn—the only caveat is that P can produce a set of plans, hence built-in function `map` has to be used to execute each element of the set at the aggregate level.

V. CASE STUDY

In this section, we illustrate how the proposed approach may be applied via simulation of a mass urban event.

A. Scenario description

Our target scenario is an urban mass event, in which part of the public runs the event application on their handheld or wearable devices. Our goal is to spot dangerous crowds as soon as possible, and selectively run a dispersal collective plan to reduce the probability of stampedes. We want our system to be able to collectively sense the density of people, to analyse such data, to plan a response and to execute such strategy. In order to build a realistic setup, we adopted the

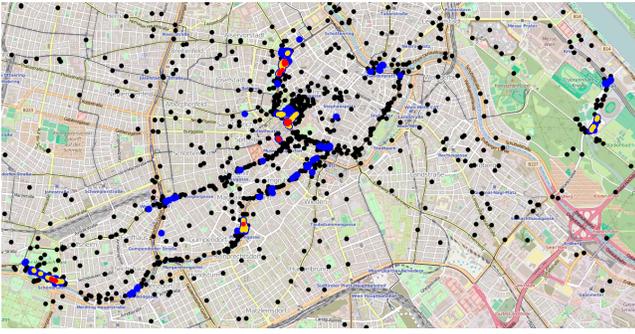


Fig. 4. A screenshot of the urban mass event simulation. Devices in high and medium danger areas are depicted in red and yellow respectively. Devices that are being suggested to actively disperse are pictured in blue. Other devices are black.

testbed used in [19]: 1479 real world traces collected during a mass city event. In this scenario, we assume that devices can communicate directly within a range of 100 meters, so that even if crowd size is high enough to disrupt the cellular network infrastructure, they still can operate. Each device begins at the initial position of its GPS trace, and from this initial situation, we let the system evolve, simulating three hours of mass event. We executed two different cases: in our control experiment, only crowd sensing was enabled; while in our experimental case, we enabled a full aggregate-MAPE, which included an emergency dispersal system as possible crowd response. Each simulated device either follows its GPS trace or is guided by the MAPE code on the device. We assume people always decide to follow the system’s safety suggestion—accounting for the possibility that people independently decide not to follow it, and study the corresponding impact, is out of the scope of this paper. Obviously, in the control simulation all nodes only follow the original GPS trace, while when the aggregate-MAPE stack is enabled, nodes can switch between the two movement strategies. The scenario has been simulated with Alchemist [20], as illustrated in Figure 4.¹

B. MAPE implementation

Figure 5 presents the core code for this scenario, relying on a number of aggregate programming building blocks [3]. The M component is responsible for estimating the number of people surrounding each device (in this case based on device communication capabilities, though more refined systems could be deployed that leverage also information coming from cameras, Bluetooth proximity sensing, etc.). The A component processes the data produced by the M component to decide whether or not a given area is dangerous. For both these stages, we used parameters taken from literature [3]. Moreover, since we only got traces for about 1500 people, but the amount of participants to the event was about 300000, we estimated the probability that a user is running the application to be 0.005, and adjusted our sensing accordingly. The P component is responsible

```
// LIBRARY
// estimates density to a circle not larger than the communication range.
def densityEstimator(p, range, footprint) {
  let nearby = unionHood PlusSelf (
    mux (nbrRange() < range)
      { nbr([getId()]) } else { [] });
  nearby.size() / p / (pi * range ^ 2 * footprint)
}
// determines whether or not the density is dangerous
def dangerousDensity(density, part, execP) {
  let s = summarize(part, (a,b)->{a+b}, 1/execP, 0);
  if (average(part, density) > 2.17 && s > 300)
    { 2 } else { 1 }
}
// returns the danger level of the area in the last minute
def crowdTracking(den, rng, execP) {
  if (recentlyTrue(den > 1.08, 60)) {
    dangerousDensity(den, S(rng, nbrRoute), execP)
  } else { 0 }
}
// MAPE SPECIFICATION
let q = 0.005; // Probability that a participant is running the app
// estimate density
let m = () -> {densityEstimator(q, 30, 0.25)};
// estimate danger level
let a = (density) -> {crowdTracking(density, 30, q)}
// disperse if within 60 meters from danger in the last 3 minutes
let p = (trk) -> {
  mux(recentlyTrue(distanceTo(trk==2) < 60, 180))
    { ["disperse"] } else { [] }}
// if there are neighboring devices in safe area, get there; else, stand still
let e = (a, label) -> {
  let mypos = self.getDevicePosition();
  env.put("goto", if(label == "disperse") {
    let dest = nbr([distanceTo(a == 2), mypos]);
    dest=mux(dest.get(0)==0){[0,mypos]}else{dest};
    maxHood PlusSelf(dest).get(1)
  } else { mypos })}
}
aggregateMAPE(m, a, p, e)
```

Fig. 5. Protelis MAPE code for crowd dispersal.

for identifying an area where the situation should get addressed: in this example, we notify all those devices that, in the last three minutes, have been within 60 meters of a dangerous area. Finally, the E component realises a simple dispersal algorithm: if the `disperse` plan is selected, each device moves toward the neighbour farthest from the danger area; if no such neighbor is found, then stand still. The idea here is that if a very big crowded area forms, the outer parts of it should disperse before the centre, to prevent the formation of dangerous waves. As shown in Figure 5, once the four components have been identified, the last instruction simply activates aggregate-MAPE. Note the ability of our framework to provide a decomposed and simple specification for each component.

C. Results and discussion

Simulation results are summarised by Figure 6. The simple algorithm deployed is sufficient to obtain a noticeable reduction of the people in dangerous areas, as well as a reduction of the number of people within 60 meters from them. The system appears to become more and more effective with time: while in the control simulation people tend to increasingly join crowds, in the version running our aggregate-MAPE this effect is widely mitigated, and the initial danger level more or less holds throughout the experiment. We can also see that a quite large number of devices get targeted for dispersion (about one fifth of the

¹Code available at: <https://bitbucket.org/danysk/experiment-2016-ecas>

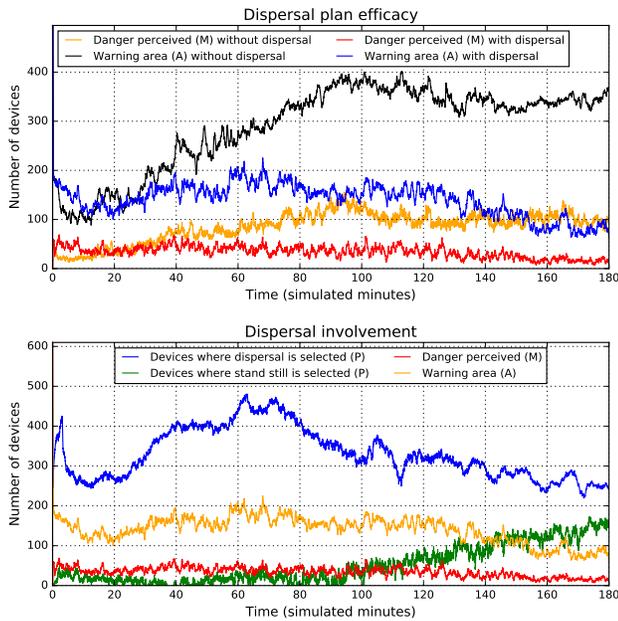


Fig. 6. Evaluation of the dispersal MAPE. The efficacy chart (top) shows how the system performs with and without the dispersal system. Despite relying on a simple algorithm, this MAPE is able to reduce both the number of devices in a dangerous density area (red and orange lines) and the number of devices within 60 meters from such areas (blue and black lines). The involvement chart (bottom) measures how many devices are selected for running the dispersal plan (blue), and how many are suggested not to move (green), as compared with the number of devices in danger (red) and close to the danger (orange).

total). This appears likely to be due to the simplistic implementation of the monitoring and dispersion algorithms, and would probably benefit from increased sophistication or parameter tuning (e.g., we used three minutes for dispersal time, but a shorter interval might produce similar results with less devices involved).

VI. CONCLUSIONS AND FUTURE WORK

Considering the need for adaptation of CASs and including it as an inner characteristic of their design offers several potential benefits, such as: (i) an appropriate abstraction level to describe complex adaptations, (ii) the possibility of having a scalable and dynamic execution environment that easily deals at runtime with different types of changes, and (iii) the generality that allows the design of solutions for a wide range of application domains. Following this principle, we have presented the aggregate-MAPE framework that—by exploiting the aggregate computing paradigm—proposes a way to model CASs as an aggregation of multiple MAPE loops. As future work we plan to experiment with aggregate-MAPE by using the full power of aggregate computing, dealing with multiple plans, and advanced sensing and actuation strategies; also, we will extend the approach considering the MAPE-K loops where the K (“Knowledge”) component will be used to share knowledge among agents so as to take more context-aware decisions; finally, we plan to further experiment the

approach in real environments as the Urban Mobility in the Smart Cities domain, also comparing performance and scalability of the aggregate approach with a conventional MAPE.

REFERENCES

- [1] F. Zambonelli, N. Biccocchi, G. Cabri, L. Leonardi, and M. Puviani, “On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles,” in *SASOW*, 2011, pp. 108–113.
- [2] A. Bucchiarone, N. Dulay, A. Lavygina, A. Marconi, H. Raik, and A. Russo, “An approach for collective adaptation in socio-technical systems,” in *IEEE SASOW 2015, Cambridge, MA, USA, September 21-25, 2015*, 2015, pp. 43–48.
- [3] J. Beal, D. Pianini, and M. Viroli, “Aggregate programming for the internet of things,” *IEEE Computer*, vol. 48, no. 9, 2015.
- [4] M. Viroli, J. Beal, F. Damiani, and D. Pianini, “Efficient engineering of complex self-organising systems by self-stabilising fields,” in *Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th International Conference on*. IEEE, Sept 2015, pp. 81–90.
- [5] M. Mamei and F. Zambonelli, “Programming pervasive and mobile computing applications: The tota approach,” *ACM Trans. on Software Engineering Methodologies*, vol. 18, no. 4, pp. 1–56, 2009.
- [6] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, “Organizing the aggregate: Languages for spatial computing,” in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, 2013, ch. 16, pp. 436–501.
- [7] F. Damiani, M. Viroli, D. Pianini, and J. Beal, “Code mobility meets self-organisation: A higher-order calculus of computational fields,” in *FORTE 2016*, ser. LNCS. Springer, 2015, vol. 9039, pp. 113–128.
- [8] IBM, “An architectural blueprint for autonomic computing.” IBM, Tech. Rep., 2006.
- [9] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing - Principles, Design and Implementation*, ser. Undergraduate Topics in Computer Science. Springer, 2013.
- [10] P. Vromant, D. Weyns, S. Malek, and J. Andersson, “On interacting control loops in self-adaptive systems,” in *SEAMS 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, 2011, pp. 202–207.
- [11] D. Weyns, S. Malek, and J. Andersson, “FORMS: unifying reference model for formal specification of distributed self-adaptive systems,” *TAAS*, vol. 7, no. 1, p. 8, 2012.
- [12] J. Bonnet, M. P. Gleizes, E. Kaddoum, S. Rainjonneau, and G. Flandin, “Multi-satellite mission planning using a self-adaptive multi-agent system,” in *IEEE SASO*, 2015, pp. 11–20.
- [13] L. Baresi, S. Guinea, and P. Saedi, “Achieving Self-adaptation through Dynamic Group Management,” in *Assurances for Self-Adaptive Systems - Principles, Models, and Techniques*, 2013, pp. 214–239.
- [14] R. Calinescu, S. Gerasimou, and A. Banks, “Self-adaptive Software with Decentralised Control Loops,” in *FASE 2015*, 2015, pp. 235–251.
- [15] M. Caporuscio, V. Grassi, M. Marzolla, and R. Mirandola, “Go-Prime: a Fully Decentralized Middleware for Utility-Aware Service Assembly,” *IEEE TSE*, vol. PrePrints, no. 1, 2015.
- [16] D. Pianini, J. Beal, and M. Viroli, “Practical aggregate programming with PROTELIS,” in *ACM Symposium on Applied Computing (SAC 2015)*, 2015.
- [17] J. Beal, M. Viroli, D. Pianini, and F. Damiani, “Self-adaptation to device distribution changes in situated computing systems,” in *IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2016)*. IEEE, 2016, to appear.
- [18] M. Viroli, D. Pianini, A. Ricci, P. Brunetti, and A. Croatti, “Multi-agent systems meet aggregate programming: Towards a notion of aggregate plan,” in *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, ser. LNCS. Springer, 2015, vol. 9387, pp. 49–64.
- [19] B. Anzengruber, D. Pianini, J. Nieminen, and A. Ferscha, “Predicting social density in mass events to prevent crowd disasters,” in *Proceedings of SocInfo 2013*, 2013, pp. 206–215.
- [20] D. Pianini, S. Montagna, and M. Viroli, “Chemical-oriented simulation of computational systems with Alchemist,” *Journal of Simulation*, 2013. [Online]. Available: <http://www.palgrave-journals.com/jos/journal/vaop/full/jos201227a.html>