Resiliency with Aggregate Computing: State of the Art and Roadmap

Mirko Viroli
Alma Mater Studiorum – Università di Bologna
Cesena, Italy
mirko.viroli@unibo.it

Jacob Beal
Raytheon BBN Technologies
Cambridge, MA, USA
jakebeal@bbn.com

One of the difficulties in engineering collective adaptive systems is the challenge of simultaneously engineering both the desired resilient behavior of the collective and the details of its implementation on individual devices. Aggregate computing simplifies this problem by separating these aspects into different layers of abstraction by means of a unifying notion of computational field and a functional computational model. We review the state of the art in aggregate computing, discuss the various resiliency properties it supports, and develop a roadmap of foundational problems still needing to be addressed in the continued development of this emerging discipline.

1 Introduction

The environment in which we all live, work, and play is increasingly saturated with computational devices, and those devices are increasingly linked with one another, with the physical environment, application services, and humans. The problems and applications of this emerging computational environment are being addressed in a wide variety of different areas, including such areas as smart cities, intelligent transportation systems, personalized health care, and the Internet of Things. A common problem in all such diverse topics, however, is to tractably engineer safe, reliable, and maintainable collective behaviors in a complex open environment comprising many devices and scales of operation.

Aggregate computing is an approach to these problems based on the recognition that many collective applications are most naturally specified in terms of aggregate properties, rather than the behavior of individual devices. For example, a crowd safety service needs to know the density and distribution of people through the environment, not the location of individuals, and users of a bike-sharing system do not typically care which bicycle or station they use as long as one is readily available nearby. Building on the natural expression of such properties in terms of collections of values spread over regions of space, called *computational fields* [26, 4], aggregate computing factors the challenging problems of programming collective adaptive systems into several abstraction layers, each of which can be engineered independently and much more tractably.

In this paper, we begin by reviewing the state of the art in aggregate computing. Following a brief discussion of the history of related work in Section 2, we present the notion of computational fields and its elaboration into the aggregate computing "stack" of abstractions in Section 3, and key results on resilience in aggregate computation systems in Section 4. We then present our view on a roadmap of foundational problems yet to be solved in Section 5 and conclude with a summary in Section 6.

2 History of Related Work

Engineering collective systems has long been a subject of interest in a wide variety of fields, from biology to robotics, networking to high-performance computing, and many more; a thorough survey of this his-

Submitted to:
FORECAST 2016

tory may be found in [4], which we summarise here. As the foundational issues of engineering collective adaptive systems remain the same, particularly when dealing with systems embedded in geometric space and having goals linked to that space (also known as spatial computers), a number of common themes have emerged across the multitude of approaches that have been developed. In particular, there are four main clusters of approaches to construct collective adaptive computational behaviours in heterogeneous networks that are identified in [4]:

- Device abstraction languages do not provide adaptivity per se, but allow a programmer to focus on adaptivity by making device interaction implicit (e.g., TOTA [26], MPI [27], NetLogo [34], Hood [38]),
- Pattern languages generally provide adaptive means for composing geometric and/or topological constructions, but little computational capability (e.g., Origami Shape Language [29], Growing Point Language [14], ASCAPE [22]),
- Information movement languages are the complement of pattern languages, providing means for summarizing from space-time regions of the environment and streaming these summaries to other regions, but little control over the patterning of that computation (e.g., TinyDB [25], Regiment [30], KQML [19]),
- General purpose spatial languages typically require more investment to use as they lack the specialisation of the other categories, but the general constructs they provide avoid the limiting constraints of the other categories (e.g., Protelis [33], Proto [3], MGS [20]).

Overall, the successes and failures of these language suggest, as observed in [5], that adaptive mechanisms are best arranged to be implicit by default, that composition of aggregate-level modules and subsystems must be simple, transparent, and result in highly predictable behaviors, and that large-scale collective adaptive systems typically require a mixture of coordination mechanisms to be deployed at different places, times, and scales. Aggregate computing is an approach that aims to draw on the successes of past approaches to produce a generalised means of programming collective adaptive systems that comply with these observations.

3 Aggregate Programming Approach

Two main lessons can be learned from previous works to more directly capture the cooperative nature of aggregates of devices. First, to raise the abstraction level, the basic mechanisms needed to achieve robust interaction of a group of devices need to be hidden "under-the-hood" of the computational model (and hence not to required to be exposed to application programmers). Second, suitable mechanisms to smoothly compose subsystems and modules are needed, in order to determine which types of coordination are appropriate between which aggregates in various regions of space and time, so as to control complex behaviours more easily.

Accordingly, the aggregate computing paradigm is grounded in three main concepts: *(i)* the reference "machine" over which collective adaptive applications run is abstracted to a conceptually single yet distributed computational device, *(ii)* the reference "elaboration process" for that machine is the manipulation of a "collective data-structure" physically distributed through part or all of the surrounding environment; and *(iii)* computation is carried out by cooperation of devices, achieving resiliency by self-organisation. This approach may then be implemented by the approach of layered abstractions depicted in Figure 1, incrementally connecting the capabilities of single devices to the development of collective adaptive applications. The remainder of this section describes each of these layers in turn.

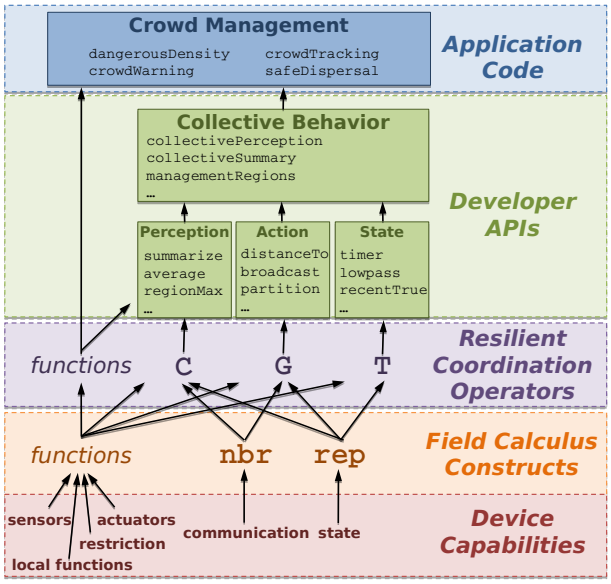


Figure 1: Layers of aggregate computing, adapted from [5]

3.1 Fields

A first key question is: what should be the shape of a "collective data-structure" manipulated with aggregate computing? Given the tight connection with physical space for the typical application scenarios we address, we consider collections of values, with each value situated in a specific point of the physical space, and at the time that value has been produced by a device, either directly by a sensor, or as result of some computation. We define the *domain* of such a collection as a set of space-time points, called *events*; note that such events correspond to computational actions (e.g., sensing, actuation, local computation) executed by some device embedded in the spatial environment. For any event ε we can thus identify a position p in space, a moment in time t, and a device d which computed it (under some coordinatisation of space and time). In line with previous works such as [26, 36, 6], we hence rely on the notion of *computational field* (or field for short), defined as a map from a domain (giving the required details of the computational environment) to some set of computational values (e.g., Booleans, numbers, or any complex computational object).

We sometimes refer to *field evolution* (instead of just a field) to emphasise how computed values evolve over both space and time, and *field snapshot* as a field over a subset of a domain selecting values at a given moment in time—i.e., selecting the last value available in each device at that time. Computational fields are a very general mechanism, useful to various purposes; one can model inputs coming from the environments as a sensor field, outputs as an actuator field, system knowledge as a data field, and any intermediate results of computation as a field of computed values. So, critically, any aggregate computation can be seen as a function from fields to fields, where input and output should have the same domain.

As the shape of a computational field has been clarified, let us consider a second key question: how does our space-time notion of collective data-structure affect the computational model? In general, computation of a value at a given event ε should depend on some contextual information, certainly including results of computations at the previous event at ε's device and information produced by sensors at ε. Additionally, some notion of local device-to-device interaction is considered. Embedded in a domain

(and depending on application-specific aspects) there is a binary notion of proximity, dictating when two devices are in the neighbouring relation. It is then assumed that computation of a value at a given event ε can depend on the value at events corresponding to the latest computation at neighbouring devices, that is, assuming a communication transferred information from a device to its neighbour. Depending on the specific computation to achieve, in particular, neighbouring devices can be restricted to consider only those belonging to a common "subdomain," identifying those devices that cooperatively bring about a common computational goal.

So to recap, computing with fields can be done by leveraging devices' ability to connect with sensors and actuators, to locally compute functions as usual and keep track of results over time, and to communicate with neighbours and possibly do so restricting the proximity relation. As shown in Figure 1 (bottom), it is on top of this lowest layer of device mechanisms, and on the notion of field, that aggregate computing grounds and builds higher-levels computing models.

3.2 Field Calculus

The field calculus (as expressed in its higher-order version in [17]) is the foundation of aggregate computing, as it provides a core language with formalised syntax, semantics and properties, on top of which more accessible programming languages can be built, and resiliency properties can be proven by construction or formal reasoning. The core idea of field calculus is to express computations by a functional language with the "everything is a field" philosophy. Given an external environment, namely a domain and sensors' values, each expression defines a field on that domain, and function application is a key ingredient that allows one to define reusable behaviour in terms of declaratively-specified transformation from fields to fields; in fact, any field computation takes field evolutions as input and produces a field evolution as output. For example, given an input of a Boolean field mapping certain devices of interest to `true`, an output field of estimated hop-by-hop distances to the nearest such device can be constructed by iterative aggregation and spreading of information, such that as the input changes the output changes to match. The field calculus succinctly captures the essence of field computations, much as λ-calculus [13] does for functional computations or FJ [21] does for object-oriented programming. Field expressions are constructed and manipulated using three syntactic program constructs:

- **Functions:** $\lambda(\mathtt{e}_1,\ldots,\mathtt{e}_n)$ applies function λ (itself an expression) to arguments $\mathtt{e}_1,\ldots,\mathtt{e}_n$, with call-by-value semantics. The function can be a "built-in" primitive (any stateless mathematical, logical, or algorithmic function, possibly in infix notation), a sensor or actuator, a function literal "$(x_1,\ldots,x_n) => \mathtt{e}$" or a user-defined function f defined as "*def* $f(x_1,\ldots,x_n)\{\mathtt{e}\}$". For instance, $1+2$ gives a flat field mapping each event to 3, *sns-temp()* the field of temperatures, $((x) => x+1)(0)$ gives 1 everywhere, and $mux(\mathtt{e}_b,\mathtt{e}_1,\mathtt{e}_2)$ computes fields out of $\mathtt{e}_b$, $\mathtt{e}_1$ and $\mathtt{e}_2$, and gives at each event the result given by $\mathtt{e}_1$ where $\mathtt{e}_b$ is `true` and $\mathtt{e}_2$ where $\mathtt{e}_b$ is `false`. Importantly, since λ can be an expression, it actually provides a field, namely, a field of functions which could change over space and time: in that case, the resulting field is obtained by preventing information flow between events where the values of λ differ. Thus, for example, $(mux(\mathtt{e}_b,+,-))(2,1)$ splits the domain in two subdomains depending on the `true`/`false` evaluation of $\mathtt{e}_b$, and computes $2+1$ in one and $2-1$ in the other.
- **Dynamics:** $rep(\mathtt{e}_0)\{\lambda\}$ defines a field holding $\mathtt{e}_0$ initially, and being updated at each event on a device by applying λ to the value held at previous event on the same device. For instance, $rep(0)\{(x) => x+1\}$ gives a field counting the number of events at each device.

- **Interaction:** *nbr*(e) gathers at each event a map from all neighbours to their latest resulting value of computing e. A special set of built-in "hood" functions can then be used to summarise such maps back to ordinary expressions. For instance, *sumHood*(*nbr*{1}) counts the number of neighbours at each event.

An example using the various constructs is the following `distance` (or `gradient`) function:

```
def distance(source){
    rep(infinity){
        (d) => mux(source, 0, minHood( nbrRange() + nbr{d}))
    }
}
```

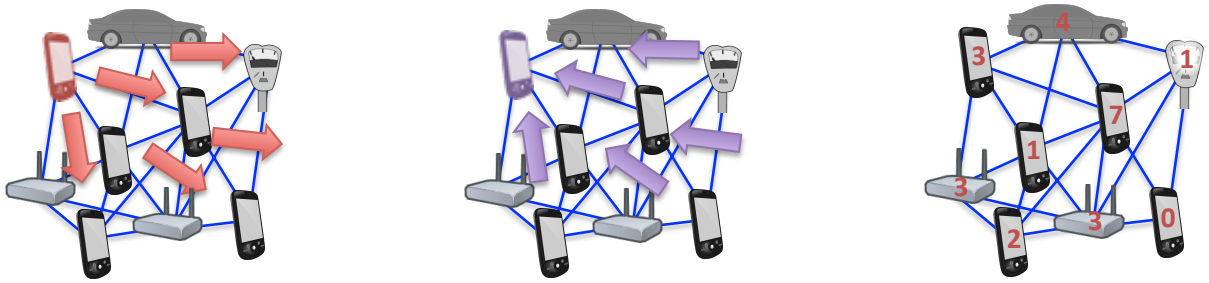
coloring field calculus keywords red, built-in functions green, and user-defined functions blue. This code estimates distance `d` to devices where `source` is `true`: it is initially infinity everywhere, and is computed over time using built-in selector `mux` to set sources to 0 and other devices by 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*).

Critically, this aggregate-level model of computation over fields can also be "compiled" into an equivalent system of local operations and message passing actually implementing the field calculus program on a distributed system [16, 17]. In particular, it defines the *computation round* behaviour, framed as a single computable function to be applied at any event.

3.3 Building Blocks and Libraries

A key advantage that aggregate programming inherits from managing computational fields functionally (as distinct from other approaches in which this is done either by diffusion/aggregation rules embedded into data items [26], or chemical-like rules embedded in "space" [37]) is that it intrinsically supports compositionality. Out of many different algorithms one can express, it is possible to factor out common behaviour into reusable functional components, all of which specify collective adaptive behaviour in terms of field-to-field transformation. As in all standard functional languages, this methodology results in the creation of complex APIs defining coherent layers of functions, where layers on top depend on layers below, raising the abstraction layer incrementally from basic ingredients to realisations of entire complex application services—see Figure 3 (top).

Most notably, experience with programming at the aggregate level and analysis of self-organisation patterns as proposed in literature (see, e.g., [18]), suggest that the three basic mechanisms one needs to ground complex applications include diffusion of information in the network as an advertisement mechanism, aggregation of distributed information as a sensing mechanism, and "evaporation" of information as a refresh mechanism. These three mechanisms can be generally supported by building blocks called G, C and T [5], whose operation is illustrated in Figure 2 and whose signatures are reported in Figure 3. As outlined in [35], different implementations can exist for these building blocks, trading smoothness and speed in different ways. More importantly, though, a whole set of library functions can be built just on top of G, C and T, by composition of these functions with one another and local functions. Figure 1 illustrates an example of the various sorts of APIs one can build, up to application services, e.g., used for large crowd management as described in [5].



(a) Operator G (b) Operator C (c) Operator T

Figure 2: "Building block" operators for distributed services: information-spreading (G), information aggregation (C), and time evolution (T), adapted from [5]

```
// Out of source, spread init value, following direction of metric, en-route applying accumulate
G(source,init,metric,accumulate)

// Gathers values of local down potential gradient, en-route accumulating and using null identity value
C(potential,accumulate,local,null)

// Locally apply decay function to initial value, until reaching floor value
T(initial,floor,decay)
```

Figure 3: Signatures of building blocks

4 Results on Resilience

Resilience may be generally defined as the ability to adapt to unexpected changes in working conditions. It is a key property for collective adaptive systems to manifest, used to ensure that system goals can be achieved even in spite of certain classes of change. Particularly important is the ability of a computing framework to provide a high degree of *inherent* resilience. This means that the system specification produced at design-time does not explicitly deal with and planning or execution of adaptation; rather, it is the underlying framework that is burdened with the goal of dynamically adapt to changes, autonomously finding ways for the system goals to be automatically achieved. Aggregate computing is a framework able to support inherent resiliency to a rather large set of changes, especially when coupled to specific techniques to structure a system specification. Technical results on resilience in aggregate computing are reviewed in this section.

4.1 Resilience to Occasional Disruption: Self-Stabilization

A typical resilience scenario is the one where a system is in good working condition until a certain occasional event from the environment creates some disruption: at that point, we wish the system to repair itself and eventually return to a good working condition. For computational fields, this notion has been formalised in [15]. A field expression e is said to *self-stabilise* if there exists a time t such that, if the environment (position and proximity of devices, and sensor fields) does not change in any

$t' > t$, then eventually, at a time $t'' > t$, the field resulting from e will no longer change either, i.e., it reaches a *self-stabilised state* (a field snapshot stable over time). Additionally, this self-stabilised state must be unique, depending solely on the stable state of the environment and on field expression e, and *not* on the field snapshot at time t. Put in another way, self-stabilisation of a field expression e implies that, given a stable state of the environment E, the resulting field necessarily eventually reaches a stable snapshot $\phi_{\mathtt{e},E}$ which solely depends on e and E: $\phi_{\mathtt{e},E}$ can be considered as the result of computation of e with "environmental condition" E. While the distance function given in the previous section enjoys this property, other similar functions are subtly not self-stabilising, like the following gossip function which keeps gossiping the minimum value of an input field across space and time:

```
def gossipMin(field){
    rep(infinity){
        (v) => min(field, minHood( nbr{v}))
    }
}
```

This is not self-stabilising because field evolution never recovers from a change in which the input field (assuming it is taken from a sensor) temporarily flips to a very low value v at some event: even after the value rises back again, v keeps being gossiped in the network.

We here refer to self-stabilisation as a notion of "resilience to occasional disruption" because the above definition implies that in a situation of continuous changes in the environment, field evolution keeps chasing a self-stabilised state, but will reach it only if there is enough time following the last change. Hence, in a situation of continuous changes, self-stabilisation per se provides no guarantee of resilience.

Self-stabilisation is undecidable in general, given that computational rounds are not even guaranteed to terminate due to the universality of even local computation. Thus, ensuring self-stabilisation is a matter of isolating fragments of the calculus that produce only self-stabilising field expressions. This problem has been addressed in [35] in which the following technical results are provided: *(i)* building blocks G, C and T are proved self-stabilising, and *(ii)* by generalising over them, a fragment of the field calculus guaranteeing self-stabilisation is identified. Most notably, such a fragment is closed under functional composition. As a result, any library or application built on top of G, C and T, and avoiding direct use of rep (like those showed in Figure 1 and in [5]), is self-stabilising by construction.

4.2 Resilience to Device Distribution: Eventual Consistency

A weakness of the above property is that the result of computation, expressed as the stabilised field snapshot, may be highly dependent on network shape. Even small perturbations to the position of a device, to the proximity relation, or to the addition/removal of a device, can make the field stabilise to a completely different result. This means that even general aspects like the overall density of devices in a given portion of space can significantly affect the result. A simple example is given by the following hop-count distance measure, estimating distances only based on the number of hops to a source:

```
def hopCountDistance(source){
    rep(infinity){
        (d) => mux(source, 0, minHood( nbr{d} + 1))
    }
}
```

There, doubling the density of devices while keeping a constant number of neighbors generally results in an increase of hop-count distances. Since practically the actual location of devices in a pervasive environment can be not known *a priori*, and even occasional changes to distribution can be the norm, one may want to introduce more specific forms of self-stabilisation, able to well tolerate changes to device distribution. Put in another way, we seek a property such that a field expression $\mathtt{e}$ necessarily eventually reaches a stable snapshot $\phi_{\mathtt{e}}$ that depends on $\mathtt{e}$, and is "mostly independent" to the shape of the environment, especially at sufficiently high densities.

The work in [8] address this issue by a notion of *eventual consistency*, essentially stating that, in addition to self-stabilisation, with the limit of event densities (devices and their work frequency) going to infinity, the stabilised state of computation converges. This notion of convergence is given by interpreting field snapshots as measurable functions over a continuous domain, and checking whether the Lebesgue integral of the absolute difference between the field snapshot obtained with a given density and that at infinite density actually converges to 0 as density goes to infinity. Though this notion does not measure the extent to which a device distribution change affects the result of computation, it can give guarantee of robustness to changes in the scale of the number of devices: at sufficient high densities, e.g. a disrupting change like increasing by 1 the order of magnitude of device densities is not going to significantly affect the shape of the stabilised field snapshot. So, one can easily expect that simpler changes like addition/removal/relocation or one or more devices will likely be irrelevant to the overall computation.

Ensuring eventual consistency is harder than simple self-stabilisation, because of a *boundary* problem. Many computations involve discrete approximations of non-continuous built-in functions (like test for equality between numbers) which tend to be very fragile to small changes in position (and distances) of devices. In [8], GPI calculus is introduced as a fragment of field calculus (a fragment significantly smaller than the one of self-stabilisation in [35]) which is based on two mechanisms. First, the only allowed form of field evolution is with a "Gradient Path Integral" construct, essentially spreading information outward from a source s and returning at each device d the result of computing the integral of a provided function across the shortest path connecting s with d. Examples of fields one can create with this construct include distance measures, broadcasts, and obstacle forecasting, all possibly realised with different kinds of metrics. Second, expressions that can lead to fragile "boundary" values (due to use of non-continuous functions) are marked, such that values cannot differ over any significant region of the field.

4.3 Resilience to Ongoing Perturbations: Controlling Dynamical Performance

What kind of resiliency support can we provide in the case of ongoing perturbations of the environment? There, it is not sufficient simply to know that a system self-stabilises, but it is very important *how* self-stabilisation is reached. Depending on the application context, we might simply seek fast self-stabilisation, while in other cases we can tolerate slow self-stabilisation provided there is smoothness, i.e., field evolution never shifts to snapshots that are too distant from the actual result of self-stablisation once reached. While fast self-stablisation can be useful with frequent, though non-continuous changes, smooth self-stabilisation may be needed with continuous changes, as in the case of many mobile networks. Two contributions have been provided in the direction of better controlling field dynamics, so far.

First, in [35] an engineering methodology is presented in which G, C and T are selectively replaced with alternative and more specialized implementations that can better trade off speed with adaptiveness in certain contexts of usage. For instance, the approach in [1] can be used to compute distances instead of by the standard implementation of G, especially when direction of movement to the source is more important than actual estimation of distance, while a multi-path collection of information can be used

instead of C's single path one when reactivity to network changes is more important than reactivity to changes in the collected data.

Second, in [32] a technique is proposed to turn gossiping into a self-stabilising process by means of running multiple replicas of gossiping in parallel at staggered times. If the proper duration of such replicas can be estimated, replicated gossip provides a much more controlled evolution of dynamics. As suggested in next section, this approach might be evaluated as a general meta-technique to improve speed and smoothness of self-stabilisation.

5 Roadmap of Foundational Problems

The results reviewed so far represent important progress in methods for the engineering of collective adaptive systems. Many foundational questions remain to be addressed, however, and resolving these questions will both broaden the applicability of aggregate computing and improve the guarantees of resilience and performance that can be made. We now present our view on the critical foundational problems still to be addressed, organizing the current key open foundational problems into four thematic groups: universality, static properties, dynamic properties, and workflow constructs.

5.1 Universality

The notion of computational universality has long been well-developed both for individual devices and for networks of devices. In this sense, there is a trivial sense in which aggregate computing can be readily shown as being universal, through the computational universality of the individual devices in the aggregate. At the aggregate level, however, studying universality helps reasoning in terms of expressiveness, allowing one to understand whether a given choice of language constructs is sufficient to express all required behaviour, and to assess comparison between different languages.

- *Discrete notion of universality.* A first notion of universality can be achieved by looking at which kinds of computations one can achieve on a given domain (defined as a finite set of events as of Section 3). Reasonable hypotheses there are that each device can compute with universal Turing power, and that inputs come from values in the local context (sensors and neighbour events).
- *Continuous notion of universality.* The work in [7, 2] suggests a different notion of universality, that focusses instead on the ability of field computations to generate fields defined over continuous space and time. Similar hypotheses here are that such fields can be locally effectively computed, and that information at an event ε can solely depend on information from the cone of past events from which ε is reachable considering a certain maximum velocity of information. With this notion, field calculus is argued to be universal in [7]. Considering less specific versions of universality is a key future work.
- *Consistency between notions universality.* Clearly, many notions of universality can be defined, and hence it will be key to compare and connect them. The work in [35] already connects discrete and continuous domains for defining the notion of eventual consistency, which can inspire the definition of a unified notion capturing both discrete and continuous domains.
- *Mobile devices.* The notions of continuous computation presented in [7, 2] address only stationary devices, while in many real-world systems the devices either move themselves or are moved by external forces (e.g., a personal device carried by its owner). Consistency between continuous and discrete computational models needs to extend to these cases, as well as accounting for the

qualitatively different behavior between tightly packed ("solid"), loosely packed ("liquid"), and sparse ("gas") distributions of mobile devices.

5.2 Static properties

A key advantage of aggregate computing compared to other approached for designing self-organising systems is its ability to compositionally and declaratively express complex behaviour. Its functional nature, in particular, allows one to readily reason formally on the expected behaviour of a program. Many interesting results have already emerged in the area of "static properties," namely, properties of the result of computation, neglecting transitory aspects that concern dynamics of evolution, but there are important areas for which these should be further extended.

- *Fragments of resilient behaviour.* As described in previous section, in [35] a fragment of self-stabilising field expression has been identified by generalisation of building blocks G, C and T into specific usage patterns for *rep* construct. Such patterns require to inspect whether certain sub-expressions enjoy properties of monotonicity, boundedness, progressiveness and so on. The work in [15] shows how automatically proving such properties in practice is not very easy. Important future work here is to find a larger fragment, with patterns easier to automatically check.
- *Beyond existing building blocks.* A reason for the current limited extent of the fragment of self-stabilising expressions is due to the fact that it originated from G, C and T, which were identified as reusable blocks even before the self-stabilisation property was established. These three building blocks allow one to functional compose operations of collection and spread of information, along with functions taking into account timing mechanisms. Although quite expressive, these do not cover all of the useful patterns of self-stabilizing algorithms. Identifying new building blocks is key to enlarge the set of resilient aggregate behaviours one can engineer. Areas of future work in this context include but are not limited to graph-based algorithms, adaptive leader election, clustering of data, flocking, and so on.
- *Model-checking and other formal methods.* The problem of addressing the foundation of group interaction in complex environments has been attacked in the community of formal models of concurrent and distributed systems, mostly by extensions of the archetype process algebra π-calculus, which models flat compositions of processes, with various versions of environment structure [10, 11, 28], shared-space abstractions [9, 36], and attribute-based ensembles [31]. In this context, rigorous formal models are typically exploited to predict quantitative and qualitative properties. To trade off verification time with accuracy, statistical model-checking [24] is often used instead of classical model-checking in addition to standard simulation, though it only partially alleviates the scalability problem. Recently, fluid flow and mean-field approximations have been proposed to turn large-scale computational systems into systems of differential equations that one could solve analytically or use to derive an evaluation of system behaviour [23, 12]. We believe that research on aggregate computing can aim at going beyond existing uses of model-checking verification techniques, relying on innovative techniques of mean-field approximation to address state-space explosion, while still allowing reasoning about aggregate processes interacting in space and time.

5.3 Dynamic properties

As discussed in Section 4, the framework of self-stabilisation, though rather expressive, does not address a number of issues of high practical impact, including performance issues as well as quantitative con-

siderations related to transitory errors in the expected behaviour. Though rather difficult to address in general, study of dynamic properties is a key ingredient for future research on aggregate computing.

- *Characterisation of resilience.* We believe that a first step towards a more clear understanding of the problem is to analyse the full spectrum of resilience, so as to identify what kind of changes an aggregate system should aim at adapting to, and the extent to which this is done in a proper and satisfactory way.
- *Speed and smoothness of self-stabilisation and eventual-consistency.* Even considering self-stabilisation, we find it key to identify formal means by which one can check, control, and then enact, various levels of speed to self-stabilisation, or of smoothness, defined as the ability of evolving towards a stable state along a trajectory guaranteeing good intermediate results. Key issues in this context include finding building block implementations for which extensive empirical analysis can be conducted to study dynamic properties, and addressing the more general problem of how properties of dynamics of certain components are preserved (or at least bounded) by composition.
- *Meta-algorithms for resiliency of dynamics.* Of great interest are those techniques that can be applied to a large class of aggregate computations that can improve their resilience, either in terms of turning non-self-stabilising computations into self-stabilising ones, speeding up self-stabilisation, or generally smoothing behaviour. Replicated instances, as initially studied in [32], are an example of such a technique, which has to be more systematically studied to identify applicability, methodologies for tuning parameters, and extensions to advance flexibility.

5.4 Workflow constructs

The functional paradigm adopted by aggregate computing promotes a clear design of the interface of piece of collective adaptive behaviour, paving the way towards composition, reuse, and substitutability. On the other hand, simple composition may itself be quickly found too limited in expressive complex interactions between modules. More generally, thinking about aggregate computations in terms of workflow (e.g., sequencing of processes) will be important for dealing with a number of complex real-world applications.

- *From fields to processes.* How might we deal with a multiset of interacting processes, as typically considered in process calculi, in the context of aggregate computing? Answering this question is key for a number of important results to be achieved, particularly for defining execution platforms for ecosystems of pervasive computing services. Possibly, this can be addressed by new constructs for the field calculus, capturing parallel composition, interleaved execution, and forms of aggregate interaction.
- *Workflow constructs.* As a notion of process is correctly identified and supported by the field calculus, new building blocks will be needed to expressively compose such processes. It will be needed to clearly identify the distributed starts and ends of a process, so as to support process sequencing, join, fork and similar workflow constructs. Likewise, virtual-machine aspects like handling of exceptions and garbage collection need to be supported in order to provide a full framework for executing complex processes at the aggregate level.

6 Conclusions

Aggregate programming is an emerging approach to the engineering of collective adaptive systems. The layered approach advocated by aggregate computing rests on the core computational model of field-based

programming embodied in field calculus. Resilience is then provided by restriction to building blocks that both provide desired resilience properties and that preserve these properties when composed with one another: to date, self-stabilization provides resilience to occasional disruptions, eventual consistency provides resilience to distribution of devices, and substitutability can be used to improve the dynamical performance of systems.

Looking toward the future, we have presented a roadmap organizing the key foundational problems for advancing aggregate programming. Beyond this roadmap, there are also a number of pragmatic challenges to address, such as improvement of aggregate programming software tools and language implementations, characterization and optimization of costs in computation, communication, energy consumption, etc, extension of the libraries and APIs, and development of additional tools and other aspects of the engineering ecosystem. Finally, ongoing work on applications will both realize the value of these approaches into real engineered systems as well as presenting challenges that we expect to feedback into new challenges for both foundational and practical investigation.

References

[1] Jacob Beal (2009): *Flexible Self-Healing Gradients*. In: *ACM Symposium on Applied Computing*, ACM, New York, NY, USA, pp. 1197–1201.

[2] Jacob Beal (2010): *A Basis Set of Operators for Space-Time Computations*. In: *Spatial Computing Workshop*. Available at: `http://www.spatial-computing.org/scw10/`.

[3] Jacob Beal & Jonathan Bachrach (2006): *Infrastructure for Engineered Emergence in Sensor/Actuator Networks*. *IEEE Intelligent Systems* 21, pp. 10–19.

[4] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli & Nikolaus Correll (2013): *Organizing the Aggregate: Languages for Spatial Computing*. In Marjan Mernik, editor: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, IGI Global, pp. 436–501, doi:10.4018/978-1-4666-2092-6.ch016.

[5] Jacob Beal, Danilo Pianini & Mirko Viroli (2015): *Aggregate Programming for the Internet of Things*. *IEEE Computer* 48(9).

[6] Jacob Beal, Kyle Usbeck & Brett Benyo (2013): *On the Evaluation of Space-Time Functions*. *The Computer Journal* 56(12), pp. 1500–1517, doi:10.1093/comjnl/bxs099. Doi: 10.1093/comjnl/bxs099.

[7] Jacob Beal, Mirko Viroli & Ferruccio Damiani (2014): *Towards a Unified Model of Spatial Computing*. In: *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France.

[8] Jacob Beal, Mirko Viroli, Danilo Pianini & Ferruccio Damiani (2016): *Self-adaptation to Device Distribution Changes in Situated Computing Systems*. In: *IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2016)*, IEEE. To appear.

[9] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto & Betti Venneri (2003): *The Klaim Project: Theory and Practice*. In: *Global Computing 2003*, *Lecture Notes in Computer Science* 2874, Springer, pp. 88–150. Available at `http://springerlink.metapress.com/openurl.asp?genre=article{&}issn=0302-9743{&}volume=2874{&}spage=88`.

[10] Luca Cardelli & Philippa Gardner (2010): *Processes in Space*. In: *6th Conference on Computability in Europe*, *Lecture Notes in Computer Science* 6158, Springer, pp. 78–87. Available at `http://dx.doi.org/10.1007/978-3-642-13962-8`.

[11] Luca Cardelli & Andrew D. Gordon (2000): *Mobile ambients*. *Theoretical Computer Science* 240(1), pp. 177–213.

[12] Luca Cardelli, Mirco Tribastone, Max Tschaikowski & Andrea Vandin (2016): *Symbolic computation of differential equivalences*. In: *POPL 2016*, pp. 137–150, doi:10.1145/2837614.2837649. Available at `http://doi.acm.org/10.1145/2837614.2837649`.

[13] Alonzo Church (1932): *A Set of Postulates for the Foundation of Logic*. *Annals of Mathematics* 33(2), pp. 346–366.

[14] Daniel Coore (1999): *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. Ph.D. thesis, MIT, Cambridge, MA, USA.

[15] Ferruccio Damiani & Mirko Viroli (2015): *Type-based Self-stabilisation for Computational Fields*. *Logical Methods in Computer Science* 11(4), pp. 1–53, doi:10.2168/LMCS-11(4:21)2015. Available at `http://www.lmcs-online.org/ojs/viewarticle.php?id=1767`.

[16] Ferruccio Damiani, Mirko Viroli & Jacob Beal (2016): *A type-sound calculus of computational fields*. *Science of Computer Programming* 117, pp. 17 – 44, doi:http://dx.doi.org/10.1016/j.scico.2015.11.005. Available at `http://www.sciencedirect.com/science/article/pii/S0167642315003573`.

[17] Ferruccio Damiani, Mirko Viroli, Danilo Pianini & Jacob Beal (2015): *Code Mobility Meets Self-organisation: A Higher-Order Calculus of Computational Fields*. In Susanne Graf & Mahesh Viswanathan, editors: *Formal Techniques for Distributed Objects, Components, and Systems*, *Lecture Notes in Computer Science* 9039, Springer International Publishing, pp. 113–128.

[18] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli & Josep Lluís Arcos (2013): *Description and composition of bio-inspired design patterns: a complete overview*. *Natural Computing* 12(1), pp. 43–67, doi:10.1007/s11047-012-9324-y. Available at `http://dx.doi.org/10.1007/s11047-012-9324-y`.

[19] Tim Finin, Richard Fritzson, Don McKay & Robin McEntire (1994): *KQML as an agent communication language*. In: *Proceedings of the third international conference on Information and knowledge management*, CIKM '94, ACM, New York, NY, USA, pp. 456–463, doi:http://doi.acm.org/10.1145/191246.191322. Available at `http://doi.acm.org/10.1145/191246.191322`.

[20] Jean-Louis Giavitto, Christophe Godin, Olivier Michel & Przemyslaw Prusinkiewicz (2002): *Computational models for integrative and developmental biology*. Technical Report 72-2002, Univerite d'Evry, LaMI.

[21] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: A Minimal Core Calculus for Java and GJ*. *ACM Transactions on Programming Languages and Systems* 23(3).

[22] M.E. Inchiosa & M.T. Parker (2002): *Overcoming design and development challenges in agent-based modeling using ASCAPE*. *Proceedings of the National Academy of Sciences of the United States of America* 99(Suppl 3), p. 7304.

[23] Diego Latella, Michele Loreti & Mieke Massink (2015): *On-the-fly PCTL fast mean-field approximated model-checking for self-organising coordination*. *Sci. Comput. Program.* 110, pp. 23–50, doi:10.1016/j.scico.2015.06.009. Available at `http://dx.doi.org/10.1016/j.scico.2015.06.009`.

[24] Axel Legay, Benot Delahaye & Saddek Bensalem (2010): *Statistical Model Checking: An Overview*. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rou, Oleg Sokolsky & Nikolai Tillmann, editors: *Runtime Verification*, *Lecture Notes in Computer Science* 6418, Springer, pp. 122–135.

[25] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin & David Culler (2002): *Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks*. In: *Workshop on Mobile Computing and Systems Applications*.

[26] Marco Mamei & Franco Zambonelli (2009): *Programming pervasive and mobile computing applications: The TOTA approach*. *ACM Trans. on Software Engineering Methodologies* 18(4), pp. 1–56, doi:http://doi.acm.org/10.1145/1538942.1538945.

[27] Message Passing Interface Forum (2009): *MPI: A Message-Passing Interface Standard Version 2.2*.

[28] Robin Milner (2006): *Pure bigraphs: Structure and dynamics*. Information and Computation 204(1), pp. 60 – 122, doi:http://dx.doi.org/10.1016/j.ic.2005.07.003. Available at http://www.sciencedirect.com/science/article/pii/S0890540105001203.

[29] Radhika Nagpal (2001): *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. Ph.D. thesis, MIT, Cambridge, MA, USA.

[30] Ryan Newton & Matt Welsh (2004): *Region Streams: Functional Macroprogramming for Sensor Networks*. In: First International Workshop on Data Management for Sensor Networks (DMSN), pp. 78–87.

[31] Rocco De Nicola, Gianluigi Ferrari, Michele Loreti & Rosario Pugliese (2013): *A Language-Based Approach to Autonomic Computing*. In: Formal Methods for Components and Objects, Lecture Notes in Computer Science 7542, pp. 25–48.

[32] Danilo Pianini, Jacob Beal & Mirko Viroli (2016): *Improving Gossip Dynamics Through Overlapping Replicates*. In Alberto Lluch Lafuente & José Proença, editors: Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, Lecture Notes in Computer Science 9686, Springer, pp. 192–207, doi:10.1007/978-3-319-39519-7_12. Available at http://dx.doi.org/10.1007/978-3-319-39519-7_12.

[33] Danilo Pianini, Mirko Viroli & Jacob Beal (2015): *Protelis: Practical Aggregate Programming*. In: ACM Symposium on Applied Computing 2015, pp. 1846–1853.

[34] E. Sklar (2007): *NetLogo, a multi-agent simulation environment*. Artificial life 13(3), pp. 303–311.

[35] Mirko Viroli, Jacob Beal, Ferruccio Damiani & Danilo Pianini (2015): *Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields*. In: IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2015), IEEE.

[36] Mirko Viroli, Matteo Casadei, Sara Montagna & Franco Zambonelli (2011): *Spatial Coordination of Pervasive Services through Chemical-inspired Tuple Spaces*. ACM Transactions on Autonomous and Adaptive Systems 6(2), pp. 14:1 – 14:24, doi:10.1145/1968513.1968517. Available at http://doi.acm.org/10.1145/1968513.1968517.

[37] Mirko Viroli, Danilo Pianini, Sara Montagna & Graeme Stevenson (2012): *Pervasive Ecosystems: a Coordination Model based on Semantic Chemistry*. In Sascha Ossowski, Paola Lecca, Chih-Cheng Hung & Jiman Hong, editors: 27th Annual ACM Symposium on Applied Computing (SAC 2012), ACM, Riva del Garda, TN, Italy, pp. 295–302.

[38] Kamin Whitehouse, Cory Sharp, Eric Brewer & David Culler (2004): *Hood: a neighborhood abstraction for sensor networks*. In: Proceedings of the 2nd international conference on Mobile systems, applications, and services, ACM Press.