Programming Actor-based Collective Adaptive Systems

Roberto Casadei and Mirko Viroli

Alma Mater Studiorum—Università di Bologna, Italy
{roby.casadei,mirko.viroli}@unibo.it

Abstract. In recent years, we are witnessing a growing interest in large-scale situated systems, such as those falling under the umbrella of pervasive computing, Cyber-Physical Systems, and the Internet of Things. The actor model is a natural choice for designing and implementing such systems, thanks to the ability of actors to address distribution, autonomy of control, and asynchronous communication: namely, it is convenient to view the *pervasive cyberspace* as an environment densely inhabited by mobile *situated actors*. But how can an actor-centric development approach be fruitfully used to engineer a complex coordination strategy, where a myriad of devices/actors performs adaptive distributed sensing/processing/acting?
Aggregate computing has been proposed as an emerging paradigm that faces this general problem by adopting a global, system-level stance, allowing to specify and functionally compose collective behaviours by operating on diffused data structures, known as "computational fields". In this paper, we develop on the idea of integrating the actor model and aggregate computing, presenting a software framework where declarative global-level system specifications are automatically turned into an underlying system of Scala/Akka actors carrying complex coordination tasks involving large sets of devices spread over the pervasive computing system.

Keywords: Aggregate Computing, Collective Adaptive Systems, Actors, Scala, Internet of Things

1 Introduction

The interplay between electronic miniaturisation and device cost reduction is leading to the diffusion into the environment of more and more elements capable of sensing, computing, and acting. This can be seen as part of the enduring goal of mankind to shape and engineer the living environment so as to make life easier or better. Nowadays, we see that a lot of opportunities arise by strengthening the connection (and mutual feedback loop) between the physical and the digital world. Multiple metaphors and buzzwords have entered the lexicon of both academia and industry that represent and evoke a similar idea, i.e., a notion of programming a "computing physical world" densely inhabited by interconnected processing units: cyberspace, service ecosystem, pervasive computing,

ubiquitous computing, Internet of Things (IoT), Cyber-Physical Systems (CPS), complex/collective adaptive systems (CAS), and so on. These scenarios challenge traditional software engineering approaches, since non-trivial complexity emerges from the combination of the multiple issues that arise, such as distribution, heterogeneity, environmental unpredictability, energetic and technological constraints, scalability, and so on.

When it comes to design and implement distributed systems composed of multiple autonomous entities, the *actor model* [2] is generally considered a primary choice, for it captures the key aspects there involved: distribution, encapsulation of control, and asynchronous communication. Indeed, several works proposed actor-based abstractions and frameworks in the context of particular distributed computing scenarios, such as wireless sensor networks (WSNs) and mobile ad-hoc networks (MANETs) [5, 21], as well as the IoT, for both middleware [20] and application [16] development. In IoT frameworks, for instance, physical devices and services can be wrapped or exposed as actors [16, 17] in order to promote application integration [17, 23], behaviour compositionality [17], and runtime adaptation [33, 28, 23].

To more stress the notion of (physical) *environment*, it is then natural to consider actors for situated systems, which we shall call *situated actors*. However, actor-based applications that involve complex coordination among several (potentially myriads of) entities, and requiring system adaptivity and resiliency, are still very difficult to build: development and maintenance tend to become convoluted and brittle due to the scattering of multiple concerns across many actor definitions and intricate conversational patterns [6]. Plausibly, this problem can be addressed by augmenting the actor model with effective abstractions for programming complex collective adaptative behaviours, providing resiliency and support for very-large scale sets of situated components somewhat inherently.

Aggregate computing [6] is a recent computational and programming model that tackles at its core the development of complex collective adaptive systems. It enables system-level behaviours to be specified as functional manipulations of whole "distributed data structures", known as *computational fields* [18, 7]—data structures consisting in a mapping from physical devices to values across time. These program specifications are then automatically translated (by a *global-to-local* mapping) into repetitive micro-level computations of individual devices. As key advantage, it becomes possible to define decentralised algorithms and coordination strategies that are independently specified from, and thus able to adapt and react to, changes in network topology, size, and density as well as to unanticipated environmental perturbations [8]. Critically, this approach is also compositional, so that coarse-grained services can be built out of simple and safe combination of smaller functional blocks, and these in turn can be combined into a service ecosystem.

In this paper, we draw a bridge between the actor model and aggregate computing, in order to establish a disciplined approach for the injection of self-adaptive and advanced coordination capabilities in complex distributed applications. On the one hand, we propose the idea of viewing aggregate computing as a

layer on top of actors that enables effective specification of complex coordination patterns. Namely, we describe an actor-based programming framework in which large sets of actors responsible of complex coordination, which we call *actor aggregates*, are programmed "in one shot" according to the aggregate computing model, to automatically and transparently interact with each other to carry on a complex computational process over space and time. With respect to traditional actor programming techniques, this approach reduces accidental complexity by fostering declarativity, separation of concerns, and modularity. On the other hand, our work suggests that a careful exposure of the actor-based view of an aggregate system can provide the means for *(i)* steering collective computation by the inputs of other non-aggregate subsystems of actors, and for *(ii)* turning the aggregate process into coordination events forwarded to the many different parts of a larger application. In a nutshell, integrating aggregate computing *on top of (as well as aside to)* actors is expected to pragmatically address open challenges in the state-of-art of IoT and CASs, by proposing a principled way to the engineering of (critical portions of) such systems.

The paper is organised as follows. First, in Section 2, we discuss the applicability of the actor model to nowadays distributed computing scenarios. In Section 3, we present a case study rooted on a CAS, and outline requirements for effectively engineering an implementation of it. Then, in Section 4, we introduce aggregate computing as an extension to the actor model that is suitable for tackling the problem of complex distributed computation and coordination. In Section 5, we present SCAFI[1], an aggregate computing framework on top of the Scala programming language, focussing on its actor-based distributed platform and then describing the language and its operational semantics through examples. Finally, in Section 6, we evaluate the idea of actor aggregates discussing an implementation of the case study.

2 Actors in the pervasive cyberspace

An *actor* [2] is a (re)active entity that represents an independent *locus of control*, encapsulates a state and behaviour, has a globally unique immutable identifier that allows for location transparency, and interacts with other actors via asynchronous message passing—each actor has a *mailbox* for message buffering. In response to a message, an actor can only (i) send a finite number of messages to other actors, (ii) create a finite number of child actors, and (iii) change its own behaviour, that is, its message processing logic. Thus, an actor system consists of an (possibly huge) evolving set of (possibly changing, mobile) autonomous actors that communicate with one another and perform some task along the way.

Actor systems very well fit highly distributed systems: since communication is based on logical identifiers, the programmer can ignore the actual physical location in which a recipient actor resides (*location transparency*). Also, actors do

[1] https://github.com/scafi/scafi

not share state, in that they communicate exclusively by exchanging messages; as a consequence, the issues related to lock-based synchronisation and mutual exclusion – as found in thread-based concurrency – are completely avoided. In addition, an asynchronous communication style better captures the way in which events occur and are perceived in the physical world [3]; anyhow, synchronicity (sometimes suitable when programming) can be supported as a particular case [2].

Given the appropriateness of actors for modelling distributed systems, it comes naturally to consider them when approaching the development of particular kinds of modern distributed systems, such as large-scale situated systems and those found in the IoT scenario [16]. Here, the *environment* abstraction becomes prominent and paves the path to *context-awareness*, namely, the ability of distinguishing situations depending on context, which is a peculiarity of any "intelligent" behaviour and is often linked to a *locality* principle, i.e., an entity is mostly directly affected by its immediate (logical or physical) surroundings (which effectively represents its context of operation). In this frame, we can introduce a notion of *situated actor* as the bridging abstraction that adds support for *situatedness* on top of plain actors. That is, a situated actor is an actor that has a given position in an environment or generally in space-time—position often inherited by the hosting or associated physical node. Concretely, such actors can be the software interface to a sensor, an actuator, a processor, or generally a computational *device* immersed in some environment, such as the urban area of a smart city, the elevator of a smart building, a room in a smart house, or an edge part of a smart appliance. In other words, after the work in [19, 34], a situated actor can be seen as an *avatar* for a physical device, and most specifically, what we can call a *space-aware avatar*.

Starting from this viewpoint, we focus on how actors could be used to implement complex decentralised behaviours, possibly involving a very-large scale set of devices (and hence actors), as those found, for example, in contexts such as crowd engineering, smart mobility, swarms of drones, environment monitoring, and so on. In principle, this would require a design of the actor system which takes into account discipline, best practices, well-known messaging [30] and structural/behavioural/reactive design patterns. However, when the logic to be expressed involves multiple concerns along different dimensions and abstraction levels, the development and maintenance processes might turn out to be very complicated and costly. There are, in fact, certain system-level properties and algorithms that are difficult to implement when reasoning in terms of individual actors and conversation patterns between actors.

What abstractions could be added to the standard actor model for addressing issues ranging from system-level adaptivity and resiliency to decentralised computation design? How could we build actor-based applications in terms of the *composition* of primitive services (e.g., reusing a crowd estimation service to develop both driving congestion-aware navigation and dispersal advice)? Following the principle of separation of concerns, we could address each problem with the more appropriate paradigm, yet importantly, recovering compatibility

between the different views so as to provide a coherent framework for building complex adaptive systems.

3 Case study: problem statement

To better present the goal of the approach proposed in this paper, we introduce a case study that will be used throughout.

Consider a mass event such as an exhibition or a concert. Suppose the event application is deployed and running in background on the smartphones of a large part of the participants, and that the location technology (e.g., the GPS on the phones) is accurate enough to estimate the order of magnitude of distances between nearby devices. Multiple services could be provided, including detection of dangerous crowd, anticipation of congestions, dispersal advice, evacuation plans, rendez-vous for groups of friends, steering to points of interest, and so on. Thus, let's consider a case study where the collection of devices has altogether to monitor – in a distributed way – the "dynamic" density of people, and react to dangerous density levels popping an alert up in each relevant smartphone. Most specifically, the service to implement could be informally expressed as follows:

(i) only devices that did not fail for at least t_{fail} seconds are considered,
(ii) partition the whole network in areas of radius r,
(iii) sense the mean density of people D_{mean} in any such area,
(iv) and when D_{mean} is greater than D_{alert}, execute some actuation A for t_{act} seconds.

The key question is: how can we *effectively engineer* such an application? The problem phrasing explicitly refers to spatial, temporal, and situation elements, hence, it would be great to specify the program logic using the same abstractions of the high-level problem description, taking a declarative stance. Ideally, we would like to leverage spatio-temporal building blocks to compose aggregate functionality into a modular solution. At the same time, the language we use should be pragmatic, intercepting static errors (e.g. concerning typing) at compile-time, guaranteeing resiliency properties of the solution, and enabling smooth reuse of existing programming structures and behaviours.

Also, critically, we would like to program such system in a way that is largely independent of the details of the underlying (networking) infrastructure. In other words, the concrete way in which interactions happen should be a platform issue, not affecting the program one writes. If an opportunistic ad-hoc network is in place, coordination may unfold by having devices exchanging messages directly with their reachable neighbours; this decentralised, peer-to-peer mode is important for scalability and for scenarios where a basic communication infrastructure is not available—as in very dense mass events. On the other hand, if the system includes one or more servers or gateways for access to the cloud, interactions and computations can be mediated by infrastructural and platform services. Taking it to a step further, one may even envision a scenario where the execution platform could also be able to transparently and dynamically adapt

the system execution strategy to *(i)* the infrastructural support available, and *(ii)* the desired quality of service.

Our goal is to implement this alerting service by an aggregation of actors running on all smartphones, interacting with the actors providing other services, e.g., to receive information sensed locally (for instance, position and esitimated distances from sources), and to communicate alert events to the actors managing smartphone screen.

4 Aggregate Computing

We note that services as those described in Section 3, as well as many other applications involving a multitude of spatially situated devices, can be best *described* (most specifically, *declaratively specified*) in terms of global, system-level properties and behaviours carried out by groups of interrelated computational elements. In other words, it is often convenient to abstract over the individual computing device and assume a more holistic stance where the "machine that computes", and hence, the machine to be programmed, is made by the (potentially large) set of devices, seen as a unique "body". Once this idea is accepted, we could start thinking about what conceptual tools we could use, what assumptions our model could be based on, and what execution platforms could be needed to address our macro specifications.

Aggregate programming [6] is an emerging paradigm that dismisses the traditional device-centric viewpoint in favour of an aggregate viewpoint where the programmable entity is the entire set of devices (also read as *actors*) that make up the system—i.e., the whole *computational fabric* mixed into the environment. Thus, by shifting the focus from the behaviour of individual actors to global patterns and evolving system structures (top-down design), the approach unburdens the programmer from the need to solve the generally intractable *bottom-up emergence engineering* problem.

The unifying abstraction that allows moving downwards (design) and upwards (execution) the micro/macro layers is that of *computational field* [18, 7, 4] (or *field* for short), which brings the concept of force field from physics into computer science. In a field ϕ, each space-time event (s, t) is mapped to a computational value $\phi(s, t)$ as computed by the device $\delta_{s,t}$ there located—or the closest to it [8]. The basic operations that are necessary to express interesting field-based computations include (i) functions mapping fields event-wise, (ii) causal transformations allowing to evolve fields while carrying on state (taking into account at each event what was the result at the event's predecessor in time), (iii) observation primitives for updating events in relation to other events in the field according to a *neighbouring relationship* in space, and (iv) domain restriction operations to isolate computations to specific portions of the fields. With such a notion at hand, fully formalised in [31], "aggregate programs" take the form of functional descriptions of field-to-field transformations over time, and one can use the compositional model of functional programming to scale with complex-

ity, and to bottom-up preserve properties of interest (e.g., self-stabilisation [11] or device location independence [32]).

We say that a field is a distributed, diffused, or global data structure because its value is given by, or emerges from, the aggregate of the values taken from an entire set of situated devices. Of course, a correspondence between the physical system and its logical representation as a field must be defined. Usually, the field directly represents the spatial situation of a deployed system, so that each networked device has a position in space, and the neighbouring relation between devices corresponds to proximity. Typically, euclidean distance with a certain threshold is used, but the neighbourhood does not strictly needs to be defined in spatial terms, nor it requires to be uniform or statically specified (fixed); the notion can be logical and ad-hoc. This notion is indeed very important: together with sensors, it concurs to define what the context is for a given element.

The execution model of aggregate computing involves partially-synchronous devices computing at discrete rounds of execution their local piece of the global program and then broadcasting to their neighbourhood the result, which we call the *export*. However, it should be stressed that we can abstract from how the interactions are actually carried out in a concrete system implementation; that is, the model is conveniently neighbour-driven, albeit its actual embodiment could involve a central coordinator or even a dynamic, hybrid infrastructure [33].

It comes natural to implement one such system using actor-based technology. In fact, an actor can logically wrap an individual device and assume a behaviour in which it iteratively accepts incoming messages from other devices and environmental events (as perceived by sensors), self-dispatches the execution of its locally compiled aggregate program, and informs its neighbour actors of the result of its computation. We shall use the term *actor aggregate* to identify the set of actors participating in an aggregate computation, that is, the actors involved in the creation of a computational field. More information about implementing aggregate computing with actors is provided in Section 5.1, where a concrete Scala/Akka framework is outlined.

Though useful, the key point of the paper is not just that aggregate computing systems can be implemented as actor systems: what is really crucial is that these two models can be fruitfully *integrated* (see Figure 1) so that each concern of distributed systems can be tackled at the right abstraction level and with the right tool.

5 Aggregate Computing with SCAFI

SCAFI (SCAla with computational FIelds) [10] is a framework that brings aggregate computing into the Scala programming language [22] and thus enables the specification of distributed and collective behaviours according to an aggregate programming style.

Note that in order to use this paradigm in practice, multiple ingredients are needed. First, the programmer needs a way – i.e., a *language* – to express aggregate computations, and there must be some *(virtual) machine* able to execute

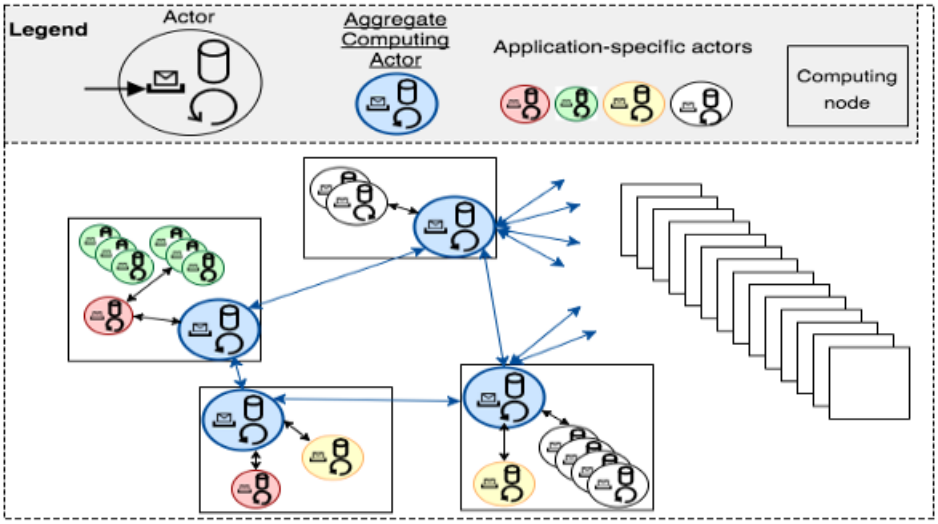

Fig. 1: Graphical representation of a generic actor-based system where application-specific actors integrate with aggregate actors so as to exploit the ability of aggregate computing to express resilient adaptive coordination strategies.

these program specifications locally in each device. Second, a distributed *system* has to be concretely specified and configured, and its runtime execution has to be supported by a *platform*. SCAFI supports all these tasks by equipping developers with a Scala-internal domain-specific language (DSL) based on the field calculus, together with an associated interpreter, and a framework upon which aggregate simulations and systems can be described and run.

Importantly, SCAFI originated with the intention of providing an integrated environment for programming aggregate systems in a modern, mainstream language. In fact, it was proposed as a type-safe and practical alternative to the dynamically typed, external DSL PROTELIS [24].

The choice to target the Scala programming language stems from both technical and practical motivations. On one hand, Scala provides a powerful static type system with type inference, an effective integration of functional and object-oriented paradigms, and advanced features that come handy for library development, also enabling the creation of fluent, DSL-flavoured APIs. That is, Scala allows us to define a standard API and thanks to both particular language constructs (e.g., by-name arguments) and syntactic sugar (e.g., curly braces can replace parentheses in method calls if a single parameter is expected), it is possible to make it appear as an "embedded" language.

On the other hand, Scala is gaining popularity especially for what concerns the development of distributed systems. The use of Akka [1] as the implementation technology for the SCAFI platform is a direct consequence of the choice of

Scala as our host language, as well as a resonable trade-off affording flexibility of design decisions and programming convenience in terms of exposed features from lower-level platform layers.

5.1 Actor-based aggregate computing platform

SCAFI comes with an actor-based distributed framework, developed in Scala/Akka, that supports both the construction and the execution of concrete aggregate systems. Though implemented with actors, it provides a convenient object-oriented façade for the configuration and management of both individual devices and whole collective applications. It also provides access to the underlying actor-based functionality, for maximum flexibility and control.

In this framework, multiple entities of the domain are modelled as actors. Sensors are essentially actors producing (a stream of) values. Conversely, actuators are actors that consume the values they are fed with, possibly with side effects. Most notably, devices become actors whose behaviour performs, at each execution round, some or all of the following duties: *(i) perception of the local context*, by reading sensors and collecting messages received from other devices; *(ii) execution of the local program obtained out of the aggregate program*, taking the local context as input and yielding both a local result and an export representing the computation just executed as output, *(iii) propagation of the export to the neighbourhood*, and *(iv) execution of actions in the local context*, by triggering actuators.

The notion of a device is a key abstraction of the aggregate computing model. However, it should be noted that the execution of computations and the awareness of neighbourhood can be moved outside the devices. In fact, the devices can ultimately be assimilated to the means by which the system perceives and acts upon specific portions of the (logical or physical) world; in other words, they are representatives of contexts of interest. Nevertheless, it is often useful to distribute as much functionality as possible to the devices, so as to leverage the locality principle and drive collective computations on a peer-to-peer interaction basis. In some cases, still, it may be necessary (or convenient) to centralise some processing or knowledge, or even to adapt the execution strategy according to the available infrastructure [33].

The instantiation of a device involves, in turn, three macro-operations:

1. `PlatformConfigurator.setupPlatform()`: creates the infrastructural support for the current physical node and yields a `PlatformFacade` instance wrapping over the newly launched Akka `ActorSystem`.
2. `PlatformFacade.newAggregateApplication()`: activates a particular application for the current physical node and yields a `SystemFacade` instance wrapping over the corresponding `AggregateApplicationActor`. Notice that one may want to run multiple applications on the same platform node.
3. `SystemFacade.newDevice()`: locally creates a logical node for the given application and yields a `DeviceManager` instance wrapping over the corresponding `DeviceActor`.

```
// STEP 1: CHOOSE INCARNATION
import scafi.incarnations.{ BasicActorP2P => Platform }
import Platform.{AggregateProgram,Settings,PlatformConfig}

// STEP 2: DEFINE AGGREGATE PROGRAM SCHEMA
class Program extends AggregateProgram with CrowdAPI {
  // Specify a "dangerous density" aggregate computation
  override def main(): Any = dangerousDensity()
}

// STEP 3: PLATFORM SETUP
val settings = Settings()
val platform = PlatformConfig.setupPlatform(settings)

// STEP 4: NODE SETUP
val sys = platform.newAggregateApplication()
val dm = sys.newDevice(id = Utils.newId(), program = Program,
                       neighbours = Utils.discoverNbrs())
val devActor = dm.actorRef // get underlying actor
```

Fig. 2: SCAFI code to locally configure the actor of an aggregate application

The actual implementation and behaviour of the above methods and actors depend on the chosen platform style and settings. Currently, two main platform styles are supported: (i) *peer-to-peer*, for fully decentralised systems where devices directly interact with each other, and (ii) *server-based*, for systems in which there is a central entity responsible for mediating device interactions and/or computations; the server might also keep a representation of the topology of the system, thus leveraging spatial features for the management of the neighbouring relationship. Critically, note that aggregate programs we shall consider in next section are completely independent of which platform (and hence, underlying communication technology) is actually used. Figure 2 shows the minimal code necessary to configure and set up a single device.

The essential steps include *(i)* the selection of the kind of platform support to use, *(ii)* the specification of the aggregate computation to run, *(iii)* the instantiation of the basic middleware infrastructure, and *(iv)* the creation of the device actor. Once a reference to the device actor is obtained, it can be used for interaction with standard actor-based functionality, of course depending on the services provided by the device actor implementation; in SCAFI, for example, device actors are observable and provide template methods for simple extension.

As pointed out in Section 4, an aggregate computing system can be seen, in a larger system, as the specific subsystem that provides advanced capabilities of robust coordination and collective computation, all with resilient adaptation to changes. In particular, actor aggregates can integrate with application-specific actors; this idea is graphically represented in Figure 1, where blue actors form an aggregate. This integration can be achieved in multiple ways: the application actors can publish values to device actors by working as sensors, or they can interact with the aggregate platform to fine-tune the neighbouring relation;

conversely, the aggregate computing actors can send messages to application actors when certain contextual conditions are met (e.g., pushing alerts to specific regulator actors when a dangerous situation is perceived)—this could be implemented by encoding the message dispatching logic directly in the aggregate program, by using "boundary" actuators, by registering application actors as observers of device actors, or by extending device actors to work as streams.

5.2 Basic syntax and semantics

The syntax and typing of the language are concisely represented by the interface (Scala `trait`) shown in Figure 3, to be inherited by any module specifying aggregate behaviour.

```
trait Constructs {
  def rep[A](init: A)(fun: (A) => A): A
  def nbr[A](expr: => A): A
  def foldhood[A](init: => A)(acc: (A,A)=>A)(expr: => A): A
  def branch[A](cond: => Boolean)(th: => A)(el: => A): A
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
  def aggregate[A](f: => A): A
}
```

Fig. 3: The Scala trait that any aggregate program has to inherit from.

Note that in Scala the return type comes at the end of a method signature; methods can be generic (with type parameters specified in square brackets); $T_i \Rightarrow T_o$ is a function type; $\Rightarrow T$ is a call-by-name parameter passed unevaluated to the function (essentially, syntactic sugar over a 0-ary function); methods can be defined to accept multiple parameter lists, and when these are single-arity, then can be specified with round or curly brackets at the call site. Critically, in the SCAFI DSL the field calculus constructs are rendered as Scala methods and hence inherit the features, the typing rules, and the syntactical constraints of Scala code.

When describing, reasoning about, or designing systems with the field calculus constructs, a twofold interpretation is possible for them. On the one hand, an operator can be seen as working on whole fields, in a sort of *global viewpoint* that also corresponds to the natural semantics of the language. On the other hand, the same operator can be considered in the context of a single device and the event at which it is executed; this is the *local viewpoint* from which the operational semantics unfolds [12]. These two perspectives are complementary and both concur to a full understanding of the model.

Aggregate programs An aggregate program consists of a set of function definitions and a body of expressions. In SCAFI, it materialises into the definition

of a "main method" of a class implementing the `Constructs` trait. The execution of an aggregate program is given by an iterative, partially synchronous "cumulative execution" of the local programs (which have the same structure and result of the aggregate program) running at the devices that comprise the system. Thus, the result emerging from an aggregate program is the field of the results achieved by executing it on each device.

Simple fields The most trivial program is a constant expression. For example,

```
"Hello, " + "World"
```

locally evaluates to the `"Hello, World"` string in every device, whereas globally it produces a uniform constant field that holds that value at any point—as obtained by concatenating the two constant fields `"Hello, "` and `"World"`, pointwise. Constant fields may be useful for expressing system parameters. Fields that are constant but not uniform could be produced by leveraging built-in or user-defined functions that behave diversely from device to device, such as:

```
mid()
```

which returns the identifier of the executing device, collectively contributing to the field of device identifiers—and similarly for any other built-in wrapping a local sensor as seen below.

Dynamic fields The `rep` operator supports the construction of fields that evolve over time, by repeatedly applying (on a round-by-round basis) a 1-ary function `fun` that maps the previous value (initially, `init`) to the new one. It enables to carry state along, from computation to computation, and use it to determine the next state. In the following example, `rep` is used for counting the number of computation rounds performed by devices since the beginning of computation:

```
// Initially 0; state is incremented at each round
rep(0){ _+1 }  // or equivalently: rep(0)( x => x+1 )
```

Note that in Scala, lambdas can be succinctly defined by using underscores "_" for denoting successive parameters. The frequency at which devices compute rounds can vary over time and from device to device, hence, in general, the resulting field will be heterogeneous in time and space. See [8] for details about field convergence.

Interaction: communication and observation Another common operation is the one used to make values of a field depend on others occurring in the neighbourhood. For the purpose, there is the `nbr` construct, which supports device interaction by means of communications carried out by broadcasting the result of

evaluating `expr` from the executing device to the corresponding neighbourhood. At each device, an `nbr` expression evaluates to a map from neighbours to the (lately received) result of evaluating `expr` at their side, effectively working as a primitive for neighbourhood observation. In the global viewpoint, a field of fields is generated, where the domain of the inner fields is the neighbourhood set for the corresponding points. Construct `nbr` has to be nested inside a `foldhood` operation, which is used to reduce the neighbour map of `expr` to a single value by accumulating on the values according to an aggregator function `acc` with identity `init`; on top of it, various specialised aggregations can be defined (e.g., `minHood` or `sumHood`). For example, the following expression

```
foldhood(0)(_+_){ nbr{1} }
```

computes the number of neighbours at each device; it works by evaluating the `nbr` expression in the context of the very current device (by actually executing the body of `nbr`) and of all its neighbours (by reading the corresponding value that has been recently communicated) and then folding over the resulting structure as you would expect from functional programming.

Similarly, in the following code we associate to each device the minimum number of computation rounds found in its neighbourhood:

```
foldhood(Int.MaxValue)(min(_,_)){ nbr{ rep(0){ _+1 } } }
```

Context-sensitiveness and sensors Every aggregate computation is locally executed against a context which includes *(i)* the export describing the previous computation (construct `rep`), *(ii)* messages received from neighbours (construct `nbr`), and *(iii)* a set of values perceived from the (physical or software) environment. Sensing is the main mechanism for extracting such values from the environment, effectively enabling context-sensitive behaviours; in practice, it is carried out by querying sensors. In the aggregate computing model there are two kinds of sensors: local sensors and neighbouring sensors.

Expression `sense[T](name)` retrieves a value of type `T` from a local sensor identified by `name`. For instance, expression

```
sense[Double]("temperature")
// Generic type for 'sense' is instantiated to 'Double'
```

returns, in any device, a `Double` value from the `temperature` sensor, which contributes to the construction of a global field of temperature readings. At this level, the manner in which sensors are defined and accessed is not specified, i.e., it is an implementation detail handled at the platform level; pragmatically, this could resolve into a sensor actor "publishing" on a particular device actor, or into the simple association of a function to a (sensor) name.

Construct `nbrvar` provides another perception feature. It is very similar to `nbr`, but instead of inspecting neighbours for an expression, it locally perceives

some information concerning neighbours—a sort of "environmental probe". This operator yields a field mapping each neighbour to some value, which has to be reduced pointwise by a `foldhood` operation as for the case of `nbr`. A commonly used environmental sensor is the one that, for each node, provides an estimate of the distance from neighbours, e.g., used as follows to retrieve the distance of the more distant neighbour:

```
// Compute the maximum distance from neighbours
foldhood(Double.MinValue)(max(_,_)){ // Also: maxHood {...}
  nbrvar[Double](NBR_RANGE_NAME)
}
```

Field domain restrictions We have seen that interaction with `nbr` is based on a notion of neighbourhood, which is defined through some physical or platform-dependent proximity relation. However, it is often useful to further regulate admissible interactions on the basis of local conditions that depend on how the computation evolved. For example, it might make sense to differentiate the algorithm carried out by devices that have recently joined the system from the behaviour of devices in a fully operating mode. In other words, it is frequently needed to define separate branches of computation that are dynamically joined by subgroups of devices. Such a feature, called *domain restriction*, is supported by the `branch` construct, which works by splitting the domain of devices into two completely isolated parts (`th` and `el`) according to a boolean field `cond` expressing some condition. Hence, the result of a `branch` expression is actually a field partitioned into two sub-fields. As an example, let's use `branch` to create a partition based on the value read from a local sensor:

```
branch(sense[Boolean]("flag")){
  compute(...)    // sub-computation
}{
  Double.MaxValue // not computing
}
```

The isolation property of field partitions requires that devices belonging to different domains are not allowed to interact. This is enforced by the execution engine, according to the operational semantics of the field calculus, through an *alignment* process. Essentially, an aggregate computation can be represented as a tree where the nodes are domain split points, and it turns out that such a structure is exactly the *export* of the computation, that is, the object that is broadcasted from any device to the corresponding neighbourhood. Then, two devices are aligned when they take the same path in the tree, i.e., when they follow the same sub-computation or partition or subfield. This works, in practice, by keeping track of the "computation path" taken by the executing device and then retaining, at any step, only the neighbour exports matching up. From this description, it should be clear that, in general, *(i)* the computation path and hence the alignment between two devices can vary from round to round, and

(ii) the alignment can be partial (i.e., can extend to a particular split point, and cannot be retrieved once lost—there is currently no primitive for domain joins).

Functions In SCAFI, there are two distinct kinds of function. First of all, there are "normal" Scala functions, which serve as units for encapsulating behaviour or algorithmic logic. For example, it is possible to define a version of `foldHood` that does not consider the device itself, or to wrap over the reading of a particular sensor with a terser linguistic item:

```
def foldhoodMinus[A](init: =>A)(acc: (A,A)=>A)(ex: =>A):A=
  foldhood(init)(acc)(mux(mid()==nbr(mid())){init}{ex})

def isSource = sense[Boolean]("source")
```

where `mux` is a purely functional multiplexer.

Second, SCAFI provides first-class aggregate functions – as defined by the higher-order version of the field calculus [12] – which have an extended semantics: they not only work as abstractions, but also as units for alignment. In other words, an aggregate function creates a computation in a domain that is identified by the function itself, so that two devices are structurally aligned only when they execute the same aggregate function. In practice, functions that have to work on whole fields should be defined by wrapping the function body with construct `aggregate`. This feature is so general that aggregate functions can be used to express the `branch` operator:

```
def branch[A](cond: => Boolean)(th: => A)(el: => A): A =
  (if(cond){
    () => aggregate{ th }
  } else {
    () => aggregate{ el }
  })()
```

Devices running distinct aggregate functions (e.g., those returned by the *then* and *else* parts of the `if` above) constitute different sub-domains and thus they are not allowed to interact via `nbr`.

As defined in detail in [12], higher-order functions are especially key to make aggregate computations open to injection of new code at run-time: special sensors can produce lambdas representing new code, which can be spread around and be executed remotely by a group of devices which then form a team carrying on a new aggregate program[2].

5.3 Composition and building blocks

In the previous subsection, the key constructs of the field calculus have been introduced: they represent the basic "moves" for expressing field-based compu-

[2] Injection of new code clearly raises security concerns which we do not address here: we simply assume that all the devices participating to the same aggregate application can be trusted.

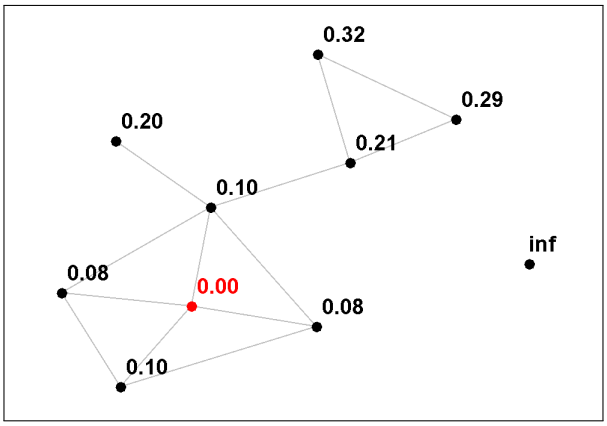

Fig. 4: Snapshot of a stabilised gradient field from a simulation in SCAFI. The bold dots represent devices (nodes). The gray lines represent connections according to the neighbouring relation. The labels above the nodes show a string representation of the result of the last computation (i.e., the root of the computation export) at the corresponding device. The red node is the source device from which the gradient is calculated.

tations. They are somewhat low-level, but the good news is that they can be combined into more meaningful patterns, for aggregate computing is designed to make any behaviour *compose* with one another functionally, and to do so at multiple levels. Thanks to this property, it is possible to envision a stack of aggregate functionality [6], i.e., layers of libraries and APIs for collective adaptive computation—from more general to more application-specific ones.

For example, one of the most common patterns is the *gradient* [14], which computes the field of the minimum estimated distances from given source points (see Figure 4):

```
def nbrRange = nbrvar[Double](NBR_RANGE_NAME)

def gradient(source: Boolean): Double =
  rep(Double.PositiveInfinity){ distance =>
    mux(source) {
      0.0
    } {
      minHood { nbr{distance} + nbrRange }
    }
```

Initially, we don't know if a path from some device to a source device exists, so an infinite distance is assumed. Source devices are obviously at a null distance from themselves. Instead, the other devices select the shortest path among those passing from the neighbourhood. The minimum distance computed is retained from round to round using `rep`.

Gradient-cast Multiple types of aggregations can be performed along a distance-gradient. In fact, it comes handy to define a generalised operator `G` as a gradient algorithm parameterised upon the `metric` for calculating increments (i.e., distances), which can carry some value of `field` from the source outward, with the logic `acc` by which such value gets evolved while ascending the gradient:

```
def G[V:OrderingFoldable](src:Boolean, field:V, acc:V=>V, metric: =>Double): V =
  rep((Double.MaxValue,field)){ case (distance,value) =>
    mux(src) {
      (0.0, field)   // ..on sources
    } {
      minHoodMinus { // minHood except myself
          (nbr{ distance } + metric, acc( nbr{ value }))
      }
    }
  }._2 // yielding the resulting field of values
```

The context bound `V:OrderingFoldable` statically enforces that the instantiated generic type `V` has an implicit `OrderingFoldable[V]` typeclass instance in scope, which provides a definition of methods `top():V`, `bottom():V`, and `compare(V,V):Int`. These constraints on `V` ensure that `minHoodMinus` can work out the minimum value for tuples `(distance,value)`, where we also assume that in the scope of the definition of `G` there are implicit `OrderingFoldables` for both `Doubles` and 2-element tuples of `OrderingFoldables`.

Upon `G`, it is straightforward, for example, to implement a basic `hopGradient` (where a distance is the number of hops from a node to another) and a `broadcast` function that simply propagates a value from source points to the rest of the network:

```
def hopGradientByG(src: Boolean): Int =
  G[Int](src, 0, acc = _+1, metric = 1)

def broadcast[V:OrderingFoldable](source: Boolean, field: V): V =
  G[V](source, field, acc = x=>x, metric = nbrRange)
```

Converge-cast Essentially, `G` allows for an information flow from source devices to their global surroundings—a sort of propagation or diffusion of values. The dual operation involves an information flow directed from a global area towards specific collection points, which can be used to perform distributed sensing. This is supported by the generalised operator `C`, which `accumulates` values along the `potential` field, starting with `local` at the sources where `potential` is maximum and aggregating while descending the chain of parents, ultimately converging to the points where `potential` is minimum.

```
def C[V:OrderingFoldable](potential: V, acc: (V,V)=>V, local: V, Null: V): V = {
  rep(local){ v =>
    acc(local, foldhood(Null)(acc){
```

```
      mux(nbr(findParent(potential)) == mid()){
        nbr(v)
      } {
        nbr(Null)
      }
    })
  }
}

def findParent[V:OrderingFoldable](potential: V): ID = {
  mux(implicitly[OrderingFoldable[V]].compare(minHood{ nbr(potential) },
                                              potential)<0 ){
    minHood{ nbr{ (potential, mid()) } }._2
  }{
    Int.MaxValue
  }
}
```

To better visualise how the algorithm works, let's consider a 3×3 grid of devices with unitary distance between rows and columns, neighbouring relation on adjacent rows and columns (i.e., Manhattan distance), and device 3 at the 2nd row and 1st column with the `"source"` sensor set to `true`. The following expression:

```
def p = distanceTo(isSource) // potential

(p, mid()+"->"+findParent(p), C[Double](p, _+_, 1, 0.0))
```

evaluates to

```
/* (1, 0->3,  3)    (2, 1->0, 2)    (3, 2->1, 1)
   (0, 3->.., 9)    (1, 4->3, 4)    (2, 5->4, 2)
   (1, 6->3,  1)    (2, 7->4, 1)    (3, 8->5, 1) */
```

as, for example, the source device (where the potential field is 0) folds (with a sum) on the aggregated values coming from the top, bottom, and right devices; conversely, "edge" devices (with no "parent") at distance `3` from the source emit the `local` value `1`.

Sparse-choice The generic operator `S` enables to select devices sparsely in such a way that the network gets partitioned into "areas of responsibility". In other words, it carries out a leader election process (see Figure 5), where `grain` is the mean distance between two leaders—according to a notion of distance expressed by `metric`. It could be implemented as follows:

```
def S(grain: Double, metric: Double): Boolean =
  breakUsingUids(randomUid, grain, metric)
```

where `randomUid` generates a random field of unique identifiers:

```
def randomUid: (Double,ID) =
  rep((Math.random()), mid()) { v => (v._1, mid()) }
```

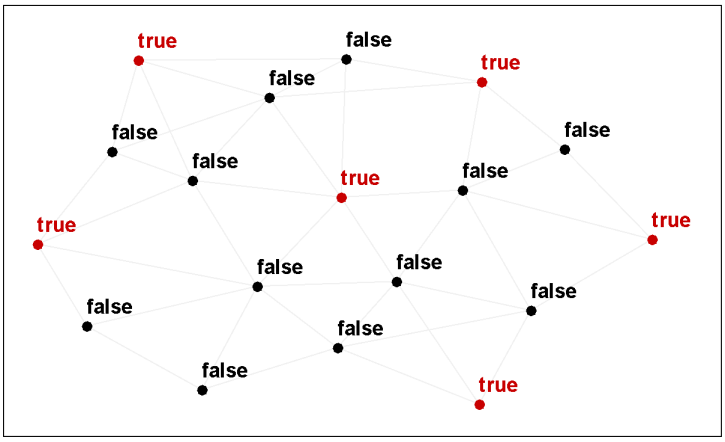

Fig. 5: Stabilised field in a SCAFI simulation for S. The red nodes are those which compute `true`, i.e., the elected leaders.

which is in turn exploited to break the network symmetry:

```
def breakUsingUids(uid: (Double,ID), grain: Double, metric: => Double): Boolean =
  uid == rep(uid) { lead:(Double,ID) =>
    val acc = (_:Double)+metric
    distanceCompetition(G[Double](uid==lead,0,acc,metric),
                        lead, uid, grain, metric)
  }
```

by means of a competition for leadership between devices defined as:

```
def distanceCompetition(d: Double,
                        lead: (Double,ID),
                        uid: (Double,ID),
                        grain: Double,
                        metric: => Double) = {
  val inf:(Double,ID) = (Double.PositiveInfinity, uid._2)
  mux(d > grain){ uid }{
    mux(d >= (0.5*grain)){ inf }{
      minHood {
        mux(nbr{d}+metric >= 0.5*grain){
          nbr{inf}
        }{
          nbr{lead}
        }
      }
    }
  }
}
```

Time-decay The `T` operator can be used to express time-related patterns, providing a convenient abstraction over `rep` construct. It works by decreasing the `initial` field with a `decay` function until a `floor` value is reached:

```
def T[V:Numeric](initial: V, floor: V, decay: V=>V): V = {
  val ev = implicitly[Numeric[V]] // getting a Numeric[V] object from the context
  rep(initial){ v =>
    ev.min(initial, ev.max(floor, decay(v)))
  }
}
```

Upon `T`, the implementation of a `timer` function is straightforward:

```
def timer[V](initial: V): V = {
  val ev = implicitly[Numeric[V]] // getting a Numeric[V] object from the context
  T(initial, ev.zero, (t:V)=>ev.minus(t, ev.one))
} // Decreases 'initial' by 1 at each round, until 0
```

In turn, `timer` supports the definition of a `limitedMemory` function that computes `value` for `timeout` and then returns `expValue` after expiration, effectively realising a finite-time memory:

```
def limitedMemory[V,T](value: V, expValue: V, timeout: T): (V,T) = {
  val ev = implicitly[Numeric[V]] // getting a Numeric[V] object from the context
  val t = timer[T](timeout)
  (mux(ev.gt(t, ev.zero)){value}{expValue}, t)
}
```

Note that the above definition of `timer` depends on the frequency of operation of a given device. If one desires a notion of temporariness that is based on physical time, it could be implemented as follows:

```
def timer(dur: Duration): Long = {
  val ct = System.nanoTime()  // Current time
  val et = ct + dur.toNanos   // Time-to-expire (bootstrap)

  rep((et, dur.toNanos)) { case (expTime,remaining) =>
    mux(remaining<=0) { (et,0) }{ (expTime, expTime - ct) }
  }._2 // Selects the component expressing remaining time
}
```

where the state about both the expiration time and the remaining time is retained across rounds via `rep`. A simulation for `timer` is shown in Figure 6.

6 Case study: implementation

We now consider the crowd density monitoring scenario outlined in Section 3, and present the aggregate program implementation (Section 6.1) and then the actor platform configuration (Section 6.2).

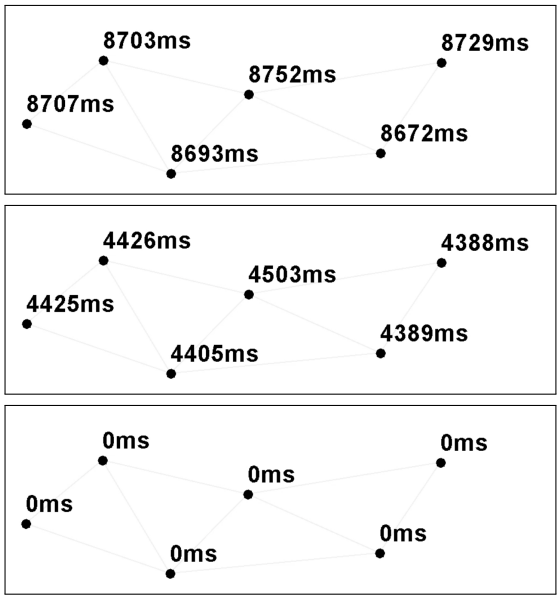

Fig. 6: Snapshots of a SCAFI simulation for `timer`; the third one depicts the stabilised field.

6.1 Aggregate program implementation

Using SCAFI, an aggregate program expressing the desired functionality can be encoded as shown in Figure 7. See Figure 8 for a set of snapshots taken from the corresponding SCAFI simulation[3].

The algorithm uses a `rep` to keep track of the last time when an alert was triggered. Then, the crowd density estimation process starts, considering only the devices that did not fail for at least a timeframe `t_fail`. The latter point is achieved with a functional block `recentlyTrue` which can be implemented as follows:

```
def recentlyTrue(dur: Duration, cond: => Boolean): Boolean =
  rep(false){ happenedRecently =>
    branch(cond){ true } { branch(!happenedRecently){ false }{ timer(dur) > 0 } }
  }
```

which returns `true` while `cond` is `true` or while the timer has not yet elapsed since `cond` flipped from `true` to `false`. In this process, the space is split into monitoring areas where the collectors of the mean density are located at a mean distance of `meanDist` (which is the double of the `radius`). If the mean density is above threshold `D_alert`, then an alert is issued. The mean density is calculated using `average`, a building block which is implemented as follows:

[3] The simulation is publicly available at the following repository: `https://bitbucket.org/metaphori/demo-aggregate-agere16`.

```
object CrowdSensingProgram extends AggregateProgram with BuildingBlocks {
  /* Parameters */
  val t_fail = (1 minute)      // Time w/o failures
  val t_act = (5 seconds)      // Time for actuation
  val D_alert = 10.0           // Mean people density threshold
  val radius = 20              // Radius of monitoring areas
  val meanDist = radius*2      // Mean distance between area leaders

  /* Program result types */
  trait Result
  case object Ok extends Result
  case object Alert extends Result
  case object FailedRecently extends Result

  trait IsActing
  case object Acts extends IsActing
  case object Idle extends IsActing

  /* Core logic */
  def main = rep( (Ok,Idle) ){ case (lastStatus, wasActing) =>
    val isWorking = recentlyTrue(t_act, lastStatus==Alert)
    val isActing = branch (isWorking) { act(); Acts } { Idle }

    (branch(withoutFailuresSince(t_fail)) {
      val areaLeader = S(grain = meanDist, metric = nbrRange)
      val D_mean = average(areaLeader, senseLocalDensity())

      // Branching working devices sensing high and low density
      branch(D_mean > D_alert){ Alert } { Ok }
    }{ // Branch of devices that have failed recently
      FailedRecently
    },
    isActing)
  }

  /* Functions */
  def senseLocalDensity() = foldhood(0)(_+_){ nbr(1) }
  def withoutFailuresSince(d: Duration) = !recentlyTrue(d, sense("failure"))
  def act[T](): Unit = { /* act in some way */ }

  /* Utility functions */
  def now = System.nanoTime()
  def never = Long.MaxValue
}
```

Fig. 7: Aggregate program in SCAFI for the crowd sensing case study.

```
def summarise(sink: Boolean,
              acc: (Double,Double)=>Double,
              local: Double,
              Null: Double): Double =
  broadcast(sink, C(distanceTo(sink), acc, local, Null))

def average(sink: Boolean, value: Double): Double =
  summarise(sink, (a,b)=>{a+b}, value, 0.0) / summarise(sink, (a,b)=>a+b, 1, 0.0)
```

where `summarise` works by first collecting the mean density at the `sink` and then propagating that value to the entire area (notice that `broadcast` refers to a "spatial broadcast" building block in the algorithm and not to an actual communication act).

6.2 Linking aggregate computing actors with other system actors

Once we came up with an aggregate program that realises our intended algorithm, this can be deployed to device actors, which will make it executing. Then, using the ideas introduced in Sections 4 and 5.1, the aggregate application can be mixed as a subsystem into a larger application, where the aggregate computing actors can coordinate and exchange information with other actors by standard message passing.

```
// PRIOR STEPS AS FROM SECTION 5
// STEP 4: NODE SETUP
val sys = platform.newAggregateApplication()
val dm = sys.newDevice(id = Utils.newId(),
                       program = CrowdSensingProgram,
                       neighbours = Utils.discoverNbrs())
val devActor = dm.actorRef // get underlying actor

// Retrieve underlying actor-system
val actorSys = sys.actorSystem
// Retrieve reference to application-specific actor on another ActorSystem
val appActor = actorSys.context.actorFor("akka://AnotherSubystem/user/SomeActor")
// Create a preprocessing actor that notifies 'appActor' only when value changes
val preprocessor = actorSys.actorOf(Props(classOf[PreprocessorActor], appActor))
// Tell 'devActor' to communicate its outputs to 'preprocessor'
devActor ! NotifyComputationResultTo(preprocessor)

// Provide a sensor value to 'devActor'
devActor ! SensorValue("temp", 27.celsius)
```

That is, by carefully exposing the underlying actor implementation, we endow aggregate systems with message-passing interfaces thorugh which it is possible to adjust, steer and enact a modularised spatio-temporal computation/coordination logic—engineered with proper tools.

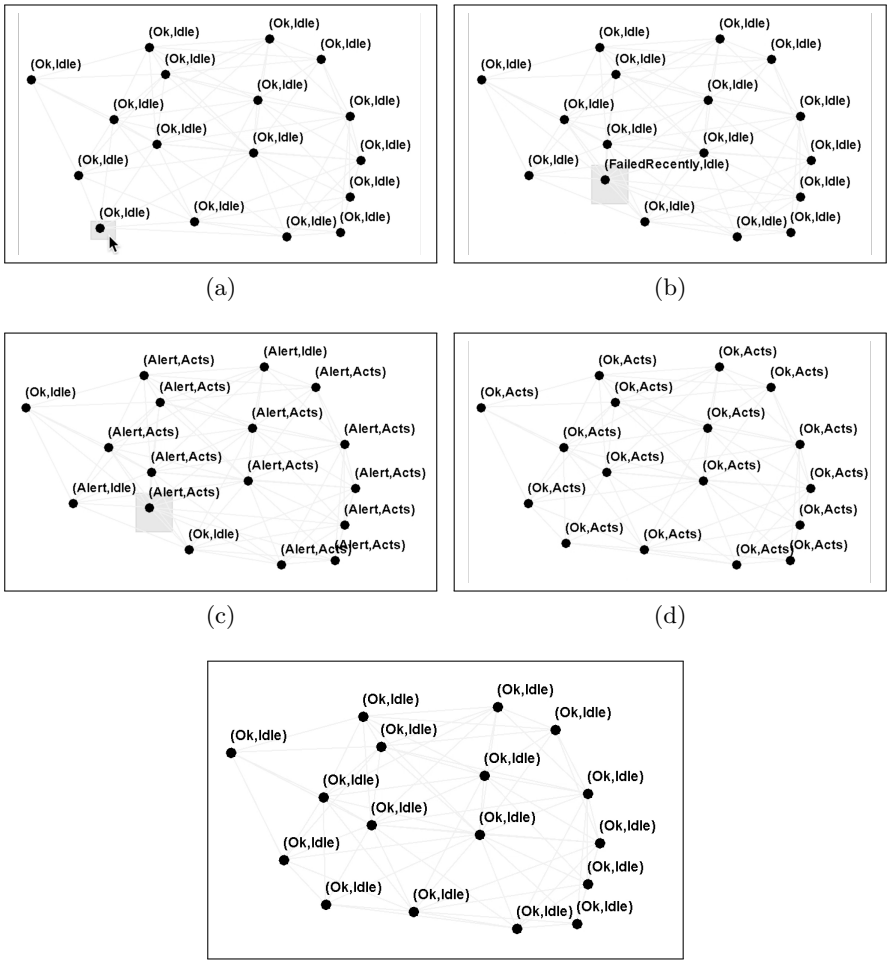

Fig. 8: Simulation of the case study in SCAFI. Snapshot 8a shows the starting network; the selected device is set to encounter a failure. In snapshot 8b the same device is moved in a crowded region. Once the device gets reconsidered for the density algorithm, as of snapshot 8c, the threshold is exceeded and an alert is spread. The device is then moved out to restore a legal density, so that the devices in snapshot 8d are not alarmed anymore but still continue to act for some time, until reaching snapshot 8e.

7 Discussion

This paper proposes an approach to the development of collective adaptive systems – e.g., as conceived in domains such as pervasive computing, IoT, or CPS – that is centered around two main ingredients: aggregate computing and actors.

Effectively engineering solutions in these scenarios is remarkably difficult, for many issues have to be dealt with in a coherent and organised fashion; in software engineering, this general challenge is usually approached incrementally by filling the abstraction gap between business requirements and technological equipment with well-engineered layers of abstractions, more likely implemented by different specialists. In other words, the problem of building end-to-end solutions is usually split in two parts: (i) the development of a platform or middleware and, on top, (ii) the construction of applications.

Specifically, the aggregate computing platform (described in Subsection 5.1) is what supports the definition as well as the runtime operation of aggregate applications. However, complex applications often consist of different kinds of services, which may be best developed with different tools. Aggregate computing, in particular, is designed to effectively engineer the self-adaptive, self-organising behaviour of a collection of situated computing devices. The key message, hence, is that the aggregate framework can be regarded as a specialised tool for effectively capturing particular needs at the application or even middleware-level, typically centered around expressing complex coordination in space-time.

On the other hands, actors have already been considered for both platform and application development in settings such as the IoT [16, 20]. Similarly, establishing an aggregate platform on top of actors appears to be congenial [33] and to promote a number of ideas related to adaptation of system execution. Indeed, the operational and network semantics of the field calculus [31], by abstracting and accounting for asynchronicity and round-by-round evolution, are well-suited to be rendered within the message-oriented and macro-step traits of the actor model. Furthermore, we argue that defining actor boundaries around aggregate computing middleware and application services paves the way to an effective integration of this framework in other platforms and more diversified applications. More concretely, for instance, field computations can be regarded as coordination events to be dispatched to actor-based handlers with glue or functional responsibilities. The idea of aggregate actors and their signified role in complex software systems may plausibly contribute, in a seamless way, to the coordination and adaptation needs encountered in pervasive computing and IoT domains.

8 Related works

An extensive survey of approaches and languages for programming aggregates of devices has been presented in [5]. In such work, four groups of approaches and their representatives – device abstraction languages (e.g., NetLogo, TOTA), pattern languages (e.g., GPL, OSL), information movement languages (e.g., TinyDB, Regiment), and general purpose spatial languages (e.g., MGS, field calculus) – are analysed with respect to a reference example involving all the essential space-time operations. In the rest of this section, we focus instead on specific aspects or similarities arising from approaches in different domains.

Our execution model, described in Section 4, might resemble a variant of the Bulk Synchronous Parallel (BSP) model [27]. In fact, aggregate computations could be carried out in a BSP-fashion. However, usually our notion of round is not global as the BSP superstep, but local to devices, and no barrier synchronisation is enforced. In fact, while BSP is a model for "precise" parallel programming also allowing accurate performance estimation, the field calculus is primarily aimed at programming distributed systems that "approximate" an ideal continuous computation. Though application-dependent, the requirements on the algorithmic actuation for complex self-organising systems such as those targeted by aggregate programming are usually less strict. In other words, the two approaches have different goals and basic assumptions. Self-stabilisation and consistency results in the field calculus are key to give meaning of global snapshots of a computational field—the interested reader can refer to [32, 8] for a deeper treatment.

The idea of computing in a global-fashion is not a prerogative of distributed programming. For instance, consider Diderot [25], a BSP-inspired parallel computing DSL for building image analysis and visualisation algorithms, e.g., in biomedical scenarios. Interestingly, it explicitly refers to spatial computation patterns; for example, it provides support for continuous tensor field manipulation, spatially-local communication via explicit neighbouring queries, and global communication as flow of information. In a language perspective, Diderot adopts an imperative style; conversely, aggregate programming relies on a functional approach, which is instrumental for achieving the compositionality of collective behaviours. Also, the demarcation arguments for BSP are still valid here, with the additional fact that Diderot is very specific in scope, mainly addressing the mathematical and parallel needs of image data processing. Nevertheless, a more detailed comparison will be conducted in the future to study additional constructs for the field calculus.

Also, the use of spatial abstractions and agent-based approaches for programming WSNs and MANETs is not new [5]. For example, SpatialViews [21] is a high-level programming model where a MANET is abstracted into *spatial views*, i.e., virtual networks of situated service provider nodes, that can be iterated on; operationally, this is achieved via migratory execution. SpatialViews works by programmatically visiting nodes and requesting services on them, in a kind of procedural orientation; again, there is a difference in abstraction, since aggregate computing aims to program an ensemble declaratively by means of space-time computation patterns. Secondly, the ability to scale with complexity via multiple spatial views and nested iterators can be contrasted to the functional compositionality which is arguably the most prominent feature of our approach.

Even though this paper has focussed on the plain Actor model, there are a number of extensions or variations to "standard actors" that could help for dealing with spatially-situated systems of mobile devices. The identification of the key features of an extended Actor model acting as the base upon which erecting an aggregate computing layer may be a useful future work. In this context, related works to be considered include, for instance, ActorSpace [9],

which provides a scoped, attribute-based mechanism that uses destination patterns for broadcasting messages to an abstract group of receivers; in particular, we could leverage this feature for implementing neighbourhood communication. Another language for programming MANETs, AmbientTalk [29], uses automatic buffering of outgoing messages for providing resilience against transient network partitions, as well as an *ambient acquaintance* mechanism for the discovery of services in nearby devices.

In the context of pervasive computing, IoT, and CPSs, several works have considered or developed extensions to actor-like models and languages in order to provide convenient design [13] and programming [26] abstractions, support the development of middleware software and platforms [20], or address peculiar problems in those scenarios (e.g., interoperability [17], dynamic functionality allocation [28, 23], and so on). For instance, ELIoT [26] is a development platform for IoT applications that provides a custom runtime fitting embedded devices as well as a dialect of the Erlang language where actor support is extended for IoT scenarios, with features including selection of diverse communication guarantees, mechanisms for remote code execution, and additional addressing schemes (e.g., for broadcasts). Calvin [23] is a platform for IoT systems that uses migratable actors for logically describing applications and enabling flexible deployment and runtime management. In [15], performance issues in IoT applications are tackled by statically defining effective actor system deployments through a cost-based method. In [28], the problem of resilient dynamic partitioning of pervasive applications has been tackled by defining these as actor systems with elastic bindings and strategies that enable the spreading of functionality actors across different devices (stretching) as well as the reverse operation (retraction), depending on the context. In the vision of swarmlets [17], actor-based accessors wrapping heterogeneous sensors, actuators, and services are proposed as a mechanism for composing functionality into well-defined components, despite technological diversity. The on-going work on aggregate computing, especially on the platform-side, does share some of the points of these efforts: the adoption of actors as a basis for the middleware layer [10] and to promote the integration of complex applications (as suggested in this paper); the opportunity for remote deployment of code at runtime [12]; actor migration as a way to transfer functionality to convenient loci and hence even adapt the system execution in a context-aware fashion [33].

9 Conclusion

In this paper, we have advanced the idea of considering aggregate computing as a layer for programming complex adaptive systems that builds on and integrates with situated actors. We have also shown how the SCAFI toolchain supports the specification of field-based aggregate computations and the construction of concrete actor-based aggregate systems. Ultimately, the case study puts into practice the field calculus constructs and building blocks by addressing a complex distributed sensing scenario.

This paper represents the first step towards the definition of a conceptual and pragmatical integration of the aggregate computing model and the actor model. This combination is proposed as a way to tackle the complexity of nowadays distributed computing scenarios (such as the IoT and CASs), where there is high need to both harness and exploit the pervasive computing cyberspace with applications exhibiting adaptation and resiliency.

Interesting future work can be envisioned to corroborate the idea along multiple directions. Extensions of the plain actor model can be taken into consideration to both support the implementation of the aggregate computing platform as well as to investigate further connections and opportunities to enrich the proposed integration; examples may include extensions to support specialised communication patterns (such as broadcasts to logical neighbourhoods), a lifecycle-oriented structuring of rounds, as well as a partitioning and relocation semantics. Among the open challenges, achieving self-adaptation of system execution – based on capabilities and trade-offs between localised computation, energy consumption, and communication overhead – is of particular relevance, spanning multi-layer architectures connecting the edge to the cloud through the fog computing, and may be approached through re-wiring and mobility of functionality actors; a related key challenge is about how to effectively support safe and efficient strategies for adaptive actor-based application partitioning [28]. Finally, we expect the idea of this paper will be better substantiated by the development of real-life applications in the context of pervasive computing and IoT; in particular, this may contribute to a deeper investigation on the use of aggregate computing components in large heterogeneous systems (emphasising the coordination and integration role of computational fields), to reveal further technical and conceptual issues and opportunities, and to provide insights by both a design and methodological perspective.

References

1. Akka. `http://akka.io`. Accessed: 2017-02-01.
2. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.
3. J. Armstrong. *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf, 2007.
4. J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.
5. J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. *CoRR*, abs/1202.5509, 2012.
6. J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 2015.
7. J. Beal and M. Viroli. Space–time programming. *Phil. Trans. of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.
8. J. Beal, M. Viroli, D. Pianini, and F. Damiani. Self-adaptation to device distribution changes. In G. Cabri, G. Picard, and N. Suri, editors, *10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2016, Augsburg,*

Germany, September 12-16, 2016, pages 60–69, 2016. Best paper of IEEE SASO 2016.
9. C. J. Callsen and G. Agha. Open heterogeneous computing in actorspace. *Journal of Parallel and Distributed Computing*, 21(3):289–300, 1994.
10. R. Casadei and M. Viroli. Towards aggregate programming in Scala. In *First Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, pages 5:1–5:7, New York, NY, USA, 2016. ACM.
11. F. Damiani and M. Viroli. Type-based self-stabilisation for computational fields. *Logical Methods in Computer Science*, 11(4):1–53, 2015.
12. F. Damiani, M. Viroli, D. Pianini, and J. Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer International Publishing, 2015.
13. B. P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling Cyber Physical Systems. *Proceeding of the IEEE*, 100(1):13–28, 2012.
14. J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.
15. A. M. Haubenwaller and K. Vandikas. Computations on the edge in the internet of things. *Procedia Computer Science*, 52:29–34, 2015.
16. R. Hiesgen, D. Charousset, and T. C. Schmidt. Embedded actors – towards distributed programming in the IoT. In *2014 IEEE Fourth International Conference on Consumer Electronics Berlin (ICCE-Berlin)*, pages 371–375. IEEE, 2014.
17. E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A Vision of Swarmlets. *IEEE Internet Computing*, 19(2):20–28, 2015.
18. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.
19. M. Mrissa, L. Médini, J.-P. Jamont, N. Le Sommer, and J. Laplace. An avatar architecture for the web of things. *IEEE Internet Computing*, 19(2):30–38, 2015.
20. A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng. Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20, 2017.
21. Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. *ACM SIGPLAN Notices*, 40(6):249–260, 2005.
22. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004.
23. P. Persson and O. Angelsmark. Calvin–merging cloud and iot. *Procedia Computer Science*, 52:210–217, 2015.
24. D. Pianini, M. Viroli, and J. Beal. Protelis: Practical aggregate programming. In *Proceedings of ACM SAC 2015*, pages 1846–1853, Salamanca, Spain, 2015. ACM.
25. J. Reppy and L. Samuels. Bulk-synchronous communication mechanisms in diderot. 2015.
26. A. Sivieri, L. Mottola, and G. Cugola. Building internet of things software with eliot. *Computer Communications*, 89:141–153, 2016.
27. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
28. J. Vallejos, E. Gonzalez Boix, E. Bainomugisha, P. Costanza, W. De Meuter, and É. Tanter. Towards Resilient Partitioning of Pervasive Computing Services.

Proceedings of the 3rd Workshop on Software Engineering for Pervasive Services (SEPS 2008), (January 2008):15–20, 2008.
29. T. Van Cutsem, E. G. Boix, C. Scholliers, A. L. Carreton, D. Harnie, K. Pinte, and W. De Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3):112–136, 2014.
30. V. Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka.* Addison-Wesley Professional, 1st edition, 2015.
31. M. Viroli, G. Audrito, F. Damiani, D. Pianini, and J. Beal. A higher-order calculus of computational fields. *CoRR*, abs/1610.08116, 2016.
32. M. Viroli, J. Beal, F. Damiani, and D. Pianini. Efficient engineering of complex self-organising systems by self-stabilising fields. In *IEEE Self-Adaptive and Self-Organizing Systems 2015*, pages 81–90. IEEE, Sept 2015.
33. M. Viroli, R. Casadei, and D. Pianini. On execution platforms for large-scale aggregate computing. In *Workshop on Collective Adaptation in Very Large Scale Ubicomp: Towards a Superorganism of Wearables*, Ubicomp '16, New York, NY, USA, 2016. ACM.
34. F. Zambonelli. Key abstractions for iot-oriented software engineering. *IEEE Software*, 34(1):38–45, 2017.