Towards Aggregate Programming in Scala

Roberto Casadei
Università di Bologna, Italy
roberto.casadei12@studio.unibo.it

Mirko Viroli
Università di Bologna, Italy
mirko.viroli@unibo.it

ABSTRACT

Recent works in the context of large-scale adaptive systems, such as those for the Internet of Things (IoT) scenario, promote aggregate programming [3], a development approach for distributed systems in which one programs the aggregate of computational devices instead of individual ones. This makes the resulting behaviour highly insensitive to network size, density, and topology, and as such, intrinsically robust to failures and changes to working conditions (e.g., location of computational load, communication technology, and computational infrastructure).

In this paper we are concerned with how this approach can impact mainstream software development, and hence outline a Scala-based support of aggregate programming, leveraging Scala advanced type system, DSL support, and actors mechanisms.

Keywords

aggregate programming; Scala; DSL; distributed platform; complex adaptive systems

1. INTRODUCTION

Building distributed systems is known to be hard in general, due to the ineluctable need to take into account issues such as concurrency, failure, consistency, and communication. The situation is then becoming harder and harder especially in recently emerging distributed computing scenarios, such as pervasive computing or IoT (Internet of Things), due to the number of computational entities, the complexity of interactions, the presence of natural limitations related to energy, communication and processing, and the tight connection with the physical world and human users—quintessential sources of entropy and unpredictability. Achieving a sound development of applications in this context, so as to ensure desired properties of robustness and scalability, calls not just for better algorithms and computing frameworks, but possibly for whole new paradigms.

PMLDC '16, July 17 2016, Rome, Italy

ISBN 978-1-4503-4775-4/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2957319.2957372

Recent research in collective adaptive software systems (CASS) has proposed *aggregate computing* [3] as a promising approach generalising over several prior models and languages addressing computations over collections of spatially-situated systems [4]. Essentially, aggregate computing allows one to express complex system-wide, global-level computations involving large sets of devices in a fully declarative way, promoting decomposition and resiliency: such specifications are then compiled back to individuals' behaviour—this approach is formally grounded in the computational fields calculus [7] and its concrete programming language named Protelis.

In order to more deeply investigate the impact of this new paradigm to software development, in this paper we report on an ongoing project to support a tighter integration of aggregate programming with mainstream programming platforms, called `scafi`[1]. This is a Scala framework that provides both an internal domain-specific language (DSL) for specifying aggregate computations via a simple API, and a platform support for the execution of distributed aggregate systems based on actors.

2. AGGREGATE PROGRAMMING

Aggregate programming [3] is a novel approach to (large-scale) distributed systems programming that supports the specification of collective behaviours in a simple, high-level, and composable way. The key idea is to shift programming from the traditional single-device viewpoint to a global viewpoint where the programmable entity is the *aggregate* body of computational elements constituting a system. This way, programmers are no longer required to solve the intricate local-to-global problem, i.e., building the desired emergent phenomenon by specifying how each component behaves and interacts with others in a fully bottom-up fashion; instead, it is possible to focus on *what* the system should exhibit, and let the computational platform define – under-the-hood – *how* the intended behaviour should be achieved by means of a coordinated set of computation and communication acts. It essentially solves the inverse, global-to-local mapping problem, though in a way that is different from more standard works such as those of choreography [13], as it is designed to work independently of the number of involved components. An immediate consequence is the independence of aggregate computations from the physical implementation details of systems, which is realised by suitably abstracting spatial distribution, topology and interaction. Most specifically, as realised by space-time programming approaches [4],

[1] http://scafi.apice.unibo.it

logical or physical neighbouring of nodes can be exploited to make interaction implicit—"I don't know *who* you are, but I know *where* you are" [18]. This notion is instrumental for the conceptual connection with systems where locality might play a major role in communication.

2.1 Computational fields

The main programming abstraction in aggregate programming is the *computational field* [7]. Generalising the notion of field in physics, a computational field (or field for short) is a function that, at a given moment in time, maps each point in space – which, by considering a networked computational system immersed in the space-time fabric, is a particular node/device – to a computational object representing the outcome of computation at that device.

The basic primitives for the manipulation of fields are defined by the computational field calculus [7]. The key insight of the approach consists in the ability to specify collective behaviors (i.e., aggregate computations) by algorithms expressed as composable functions manipulating fields. In other words, aggregate computations are represented by a declarative specification of functional operations involving fields, though, under-the-hood, they are turned into repetitive, gossip-like interactions between individual devices.

This approach is shown to support a solid programming methodology in which composable and reusable high-level library components of aggregate behaviour can be defined. Most notably, a number of resilient coordination operators have been identified [16] on top of the field calculus; these operators capture recurrent self-organisation patterns [9] and also exhibit useful properties, such as self-stabilisation [17].

2.2 Computational Model

An aggregate system consists of a (possibly large) number of computational devices (also called *nodes*, as a system can often be seen as a network of elements; *things*, as in the IoT interpretation; or *points*, as in a space), all executing the same aggregate program at asynchronous rounds of computation. According to contextual information (e.g, sensor values or neighbourhood data), different devices may take different branches of computation, i.e., computing sub-fields in different domains of execution. The state of the whole system, thus, can be represented as the field of values computed at each device. Interaction depends on a notion of locality, i.e., a device can only directly communicate to all its neighbours, as defined by an application-specific proximity relation. The communication is carried out by repeatedly broadcasting the latest computed state (called an *export*) to the entire neighbourhood: the shape of this state, and how it affects and is affected by computations, is precisely defined by our language semantics [7], as described in the following.

In each device, a computation round consists of the following steps:

1. *Creation of the execution context state*—which includes the latest computed local value, the most recent exports received from neighbours, and a snapshot of local sensor values.

2. *Local execution of the aggregate program*, which, based int context state, yields the new state (or export).

3. *Propagation of the computed export to the entire neighbourhood*, done by a broadcast.

4. *Activation of the actuators*, with the input given by the result of computation.

How can the system exhibit the intended global behaviour if every device is given the same aggregate program specification? Intuitively, the computational model embodied by the field calculus is able to make sense of the context-sensitiveness and the different local conditions (*deltas*) at the device-site and hence to construct and constrain the global phenomenon.

3. AGGREGATE PROGRAMMING IN `scafi`

For what concerns development support, aggregate programming is currently provided by Protelis [15], an external DSL developed in Xtext and hosted in Java, integrating with the biochemically-inspired Alchemist simulator [14]. In the attempt to provide a more integrated development toolchain aimed at the construction of actual systems, a new aggregate programming framework, called `scafi` (Scala Fields), is under construction.

`scafi` is a framework that consists of two main pieces: *(i) aggregate programming support*, by a Scala-internal DSL that provides a syntax and the semantics for the basic constructs of the computational field calculus, by which aggregate computations are naturally expressed and combined with standard code; and *(ii) aggregate platform support*, allowing configuration and execution of distributed aggregate systems.

The rationale for the choice of Scala as the host language for building an aggregate programming platform is both technical and pragmatic. Scala is a modern language, interoperable with Java, which integrates the object-oriented and functional paradigms in a seamless way, has a powerful and expressive type system, and offers advanced features supporting the design of high-level software libraries and fluent APIs. In particular, Scala combines the advantages of a static type checking with features allowing for productivity and syntactical conciseness, such as: type inference, implicits, generic and functional programming features, and syntactic sugar. This allows library designers to design and implement APIs with a DSL-like flavour, which do appear to users as "embedded languages." Additionally, Scala also provides standard libraries and community support for distributed systems engineering, with toolkits like the Akka actor framework.

3.1 The basic field calculus constructs

Syntax and typing of the basic primitives of the field calculus are declared by the methods of the `Constructs` trait, and follow the formalisation in [8]:

```
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 aggregate[A](f: => A): A
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

The field calculus interpreter provided with `scafi` implements the above interface, which is also mixed-in any API of aggregate programs. In Scala, methods can have multiple parameter lists, type parameters in square brackets,

and by-name parameters (specified via $\Rightarrow T$), which are passed unevaluated and get calculated every time they are referenced. Function types take form $(T_{11}, ..., T_{1n}) \Rightarrow ... \Rightarrow (T_{k1}, ..., T_{km}) \Rightarrow R$, where T_{ij} is the type of i-th parameter of j-th parameter list, and R is the return value type.

The field calculus constructs can be explained according to two complementary viewpoints:

1. *Local viewpoint* – it corresponds to the operational semantics and refers to the conventional device-centric interpretation where aggregate computations are considered in the context of a single device: there, a value is the result of computation in a node at a given time.
2. *Global viewpoint* – it corresponds to the natural semantics and refers to the aggregate-level interpretation of programs as manipulations of dynamic computational fields: there, a value is a system-wide snapshot of a computational field at a given time.

A `scafi` aggregate program consists of a set of function definitions and a body of variable declarations and expressions, all of which can mix core field constructs with any Scala existing library. A result of a program is the field of the values obtained by evaluating its last expression. The most trivial expression is one that simply evaluates to a constant value, such as a boolean, a number or a string. For instance, value

```
"Hello, World"
```

should be interpreted as a constant field that holds that value at any point; this means that the result of the local computation of every device will be the `"Hello, World"` string.

Construct `sense` is used for reading a value from a local sensor. Expression

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

yields, in any device, a `Double` value from the `temperature` sensor, effectively producing a field of temperature reads. Sensing is important because it enables for context-sensitive behaviours, and is the main mechanism for extracting inputs from the environment. The concrete access to the sensor is handled at the platform level, using the appropriate technology. The programmer can assume that the execution context for a round includes a map from sensor names to the corresponding values, e.g., as obtained by sampling sensors before computation.

The `rep` construct allows to build a field that evolves over time, round by round, by starting from `init` and repeatedly applying a state-transformation function `fun` that expresses the intended dynamics. A simple example involves counting the number of computational rounds performed by a device since the beginning of computation:

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

where we use curly brackets instead of parentheses (they are interchangeable) for the second parameter list, whose argument is a 1-ary function expressed via the shorthand lambda syntax, also using underscores _ for denoting parameters (each occurrence of the symbol denotes a successive parameter). As the frequency at which devices compute rounds (and hence send messages) to neighbours can vary over time and from agent to agent, the resulting field will change over time, from 0 everywhere, to numeric integer values increasing over space and time heterogeneously. In fact, the aggregate computing model generally assumes partial synchronicity [1], though in most cases even full asynchrony of rounds can be assumed [6].

The `nbr` construct supports interaction by means of bidirectional communication between a device and its neighbourhood. The neighbourhood (of a node) consists of the set of devices included by the neighbouring relation, which is often the euclidean distance over a metric space, as to reflect the physical analogue of devices immersed in an environment such as that of a smart building or a urban area in a smart city—still, the underlying platform may handle the neighbouring relationship in various ways. The result of a `nbr` operation is a map from neighbours to the result of evaluating the argument at their side; in other words, it works as a primitive for neighbourhood observation. Construct `nbr` has to be nested inside a `foldhood` operation, which essentially reduces one such map (`expr`) to a single value by applying an aggregator function `acc` with identity `init`; on top of it, derived operations such as `minHood` or `sumHood` can be defined. Consider the following examples:

```
// Counting number of neighbours at each device
foldhood(0)(_+_){ nbr{1} } // sum 1 across neighbours

// Is sensor "sns" active in every neighbour?
foldhood(true)(_&&_){ nbr{ sense[Boolean]("sns") } }
```

Locally, `foldhood` works by evaluating `nbr` in the context of the very current device (by executing the expression passed along to `nbr`) and of all its neighbours (by reading the corresponding value that has been recently communicated) and then operating as the conventional fold function of functional programming on the resulting structure. The first example sums value 1 computed in each neighbour, hence the resulting field maps each device to the result of counting its number of neighbours; the second example is the field mapping each device to a boolean value, holding true if any neighbour has sensor `sns` holding itself true.

Another perception feature is provided through the `nbrvar` construct. This operator is used to query a neighbouring sensor, which is a kind of "environmental probe" that produces a field mapping each neighbour to some value. Just like `nbr`, the result of `nbrvar` have to be reduced pointwise by a `foldhood` operation. The typical example of environmental sensor is one that estimates, for each node, the distances of neighbours:

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

// Compute the maximum distance of a neighbour
foldhood(Double.MinValue)(max(_,_)){ nbrRange() }
// equivalently: maxHood{ nbrRange() }
```

Up to now, we have seen that interaction is achieved by observing neighbours, according to a platform-dependent proximity relation. However, it is often useful to further regulate admissible interactions by defining separate branches of computation, dynamically assigned to subgroups of devices. This feature, called *domain restriction*, is supported by the `branch` construct, which splits the domain of devices into two parts according to a boolean field `cond` expressing some condition. Each of the two parts compute a different sub-field in complete isolation, `th` and `el`. The execution engine (via field calculus' operational semantics) ensures that devices belonging to one partition compute the same

branch expression—this process is known as *alignment*, and we say two devices are aligned when evaluating a given sub-expression. Dually, the engine also ensures that devices in separate partitions are restricted compute the two branches without interactions between them. Note that, in general, (i) the alignment between two devices can vary from round to round, and that (ii) the selection of a branch happens in the scope of evaluating a certain sub-expression—elsewhere interaction is generally possible. As an example of `branch`, consider the case we want to execute some aggregate computation only on a subset of the devices of the network, while the complementary subset must not participate; for the purpose, we have to create a domain partition:

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

Note that both the *then* expression and the *else* expression are thunks (i.e., they are passed non-evaluated to `branch`). `branch` behaves much like the typical `if` construct of programming languages, except that it also deals with alignment.

Finally, `scafi` supports an higher-order version of the field calculus [8] which introduces first-class aggregate functions. Construct `aggregate` is used to wrap a function body that should work on whole fields. Essentially, this operator labels a function so that it can be used to define a whole new computation from fields to fields: when such functions are combined so as to create fields of functions, then alignment is based on the function identity, and devices are structurally aligned only when they execute the same aggregate function. As an example, let's consider an implementation of `branch` that uses `aggregate`:

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

where `mux` is a purely functional multiplexer. The devices running the same aggregate function (e.g., the one returned by the *then* part of the `mux`) constitute a partition, i.e., they can interact to each other via `nbr` only within that restriction of the domain.

3.2 From primitives to building blocks

The primitives of the field calculus form a minimal set of moves capturing distributed, situated field-based computations. As such, they are somewhat low-level, and one easily wants to raise the abstraction level to better reason about complex collective adaptive systems. The intrinsic composability of the functional paradigm fosters a systematic factorisation of behaviour into reusable building blocks out of the basic constructs, and a coherent set of aggregate behaviours can then be packaged into a software library. In `scafi`, which is a fully featured function framework, generic parts of aggregate program logic can be encapsulated into standard Scala functions. As an example, consider the definition of a function `foldhoodMinus`, which behaves as `foldhood` except that it excludes the device itself from the evaluation of `expr`.

```
// mid is a special sensor yielding the device unique id
def isMe = nbr{ mid() } == mid()

def foldhoodMinus[A](init: => A)
                    (aggr: (A, A) => A)
                    (expr: => A): A =
  foldhood(init)(aggr){ mux(isMe){ init }{ expr } }
```

The paradigmatic example of computational field is known as gradient [9]. In a gradient field, each node computes its distance (e.g., hop-by-hop or estimated) from the nearest node where a `source` field holds a true value:

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

Here is how it works: starting from an estimated distance `d` that is infinity everywhere, on source nodes we compute distance 0; the other devices compute the minimum value among the neighbours' value increased by node-to-neighbour distance; immediate neighbours to a source node, hence, compute their distance to that node (if that is the nearest one), and iteratively, all others get progressively updated. In the absence of network changes, this process ultimately stabilises into a field of values that are increasing along the distance to source nodes and are indeed an estimation of that distance. Moreover and most notably, the algorithm is able to adapt to network change (which may be due to node mobility or temporary environmental conditions affecting the neighbouring relation) and it does so by refreshing (in the `rep`) the computed distances from the change point.

The gradient algorithm is a very general one and finds wide application. By exploiting similarly created library functions such as `managementRegions` (partitioning the network), `average` (distributed sensing), and the like [3], one can elegantly express very complex behaviour, like a whole crowd sensing service. E.g., according to [10], the following function checks whether in a region with radius `r` there are at least `n` people experiencing high density of crowd (2.17 people/square meter):

```
def crowd(r: Double, n: Int): Boolean = {
  val mr = managementRegions(grain = r*2)
  average(mr, localDensityEstimation()) > 2.17 &&
    summarize(mr, (_:Double)+(_:Double), 1) > n
}
```

where `grain=r*2` is used to make explicit the parameter name for documentation purposes. Note (i) the degree to which the provided specification completely abstracts from individual node's behaviour and network shape, and (ii) how a useful application is declaratively built out of composable and reusable pieces of behaviour.

3.3 Generic building blocks

We now see how to approach the development of layers of building blocks, from more low-level/general to more high-level/specific. A first step is to identify useful recurrent combinations of the primitives and abstract them into functions. In addition, by capturing common patterns of distributed computation and giving them a name, we can provide some guidance for the design of aggregate behaviour. Research in the context of bio-inspired computing has provided significant results in this direction, by determining a handful of self-organisation mechanisms [9] where functional and non-functional properties at the system-level are achieved by means of decentralised, local interactions among a (large) number of (simple) agents.

Moreover, there are some properties that we would like our distributed algorithms to exhibit. One such property is *self-stabilisation* [17], which ensures that a system, inde-

pendently of the current state, will eventually reach a stable state in finite time that is not affected by transitory events. Non-functional properties such as robustness and response time are also very important and, especially in large-scale systems, can make the difference between an application that works and one that does not.

Also prominent is how these properties relate to composability: are they maintained by composition? If not, how are they affected? Which combinations do conserve them? In particular, it has been proved that self-stabilisation is preserved by composition [17], i.e., it is formally guaranteed that composite structures of self-stabilising algorithms also self-stabilise. This result is significant because it means that the benefits of low-level components propagate to combinations of these up to higher levels and application-specific logic.

An initial set of general self-stabilising coordination operators has been identified in [16]. Most notably, these include:

- **G**radient-cast: accumulates values "outward" along a gradient starting from source nodes.
- **C**onverge-cast: collects data distributed across space "inward" by accumulating values from edge nodes to sink nodes down a "potential" field.
- **T**ime-decay: supports information summarisation across time.

```
def G[V:OB](src: Boolean, init: V,
            acc: V=>V, metric: =>Double): V

def C[V:OB](potential: V, acc: (V,V)=>V,
            local: V, Null: V): V

def T[V:Numeric](initial: V, floor: V, decay: V=>V): V
```

Next, we introduce one of such operators, `G`, explain its details and show how it can be used to define other building blocks supporting non-trivial distributed algorithms.

3.3.1 Gradient-cast

The `G` operator subsumes and generalises the `gradient` functionality. It works by simultaneously performing two tasks: i) construction of a distance-gradient from sources (`src`) according to a `metric` expressing node-to-node increments, and ii) accumulation of values (via `acc`) along the gradient starting from `init` at the `src` points. In `scafi`, it can be encoded as follows:

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

where

- Syntax `(a:A,b:B)` is a shorthand syntax for a 2-element tuple (i.e., a `Tuple2[A,B]` instance), whose elements can accessed via getters `_1` and `_2`, respectively.
- Syntax `V:OB` espresses a particular constraint, called *context bound*, on the type parameter `V`. It is a succint alternative to extending the function signature with an implicit parameter [12] of type `OB[V]`. An implicit parameter is one that can be provided – in addition to the usual explicit manner by the programmer – implicitly or automatically by the compiler.
- The `OB[A]` type class (where `OB` stands for *Ordering, Bounded*) defines a bounded and totally ordered representation for a type `A`:

```
trait OB[A] {
  def top: A
  def bottom: A
  def compare(a: A, b: A): Int
  def min(a:A, b:A): A = if(compare(a,b)<=0) a else b
  def max(a:A, b:A): A = if(compare(a,b)>=0) a else b
}
// Classes/traits admit same-name companion objects
object OB { // Singleton object
  implicit val of_d = new OB[Double] {
    def top: Double = Double.MaxValue
    def bottom: Double = Double.MinValue
    def compare(a:Double, b:Double) = (a-b).signum
  }
  // others...
}
```

Thus, in the above definition of `G`, an (implicit or explicit) instance of the `OB` type class must be available for generic type `V`, so that `minHoodMinus[T:OB]` can work out the minimum `T`-expression value on the neighbourhood (by convention, `*hoodMinus` operators exclude the device itself from the neighbours set). This must also be true for `Tuple2[+A,+B]`, given that an `OB` exists for both `A` and `B`.

3.3.2 From `G` to `channel`

The goal of a channel algorithm is to draw a link from a source area to a destination area. In other words, a channel is a field that holds `true` for nodes located in the path from a starting point to a target point, and `false` anywhere else. More generally, each device has to compute an estimate of the distance from itself to the unit-wide path, which expresses the degree to which that node does not belong to the channel. Now we see an implementation in `scafi` of `channel` that builds on `G`-derived operators.

Using `G`, it is easy to implement a `broadcast` operation to propagate a value across the whole network of devices:

```
def broadcast[V:OB](source: Boolean, init: V): V =
  G[V](source, init, x=>x, nbrRange())
```

as well as a functional block `distanceTo` that computes, in each node, the distance between that node and a source point:

```
def distanceTo(source: Boolean): Double =
  G[Double](source, 0, _ + nbrRange(), nbrRange())
```

Finally, once we have a way to spread everywhere the distance between the `source` and the `target`:

```
def distBetween(source: Boolean, target: Boolean): Double =
  broadcast(source, distanceTo(target))
```

we have all the ingredients for a `width`-wide `channel` operator connecting a source and a destination:

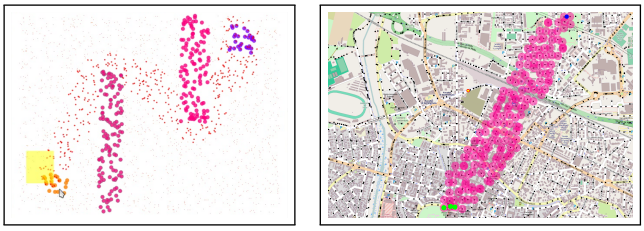

Figure 1: Self-stabilising channel at work (with Alchemist simulator [14]): automatically bypassing obstacles (left), and smoothly adapting to a smartcity scenario. (right)

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

def channel(src: Boolean, dest: Boolean, width: Double) =
  distanceTo(src) + distanceTo(dest) <=
    distBetween(src, dest) + width

channel(isSource, isTarget)
```

Note the degree to which the implementation of `channel` looks like a global specification and how it naturally comes out from composable high-level operators. Moreover, this algorithm self-stabilise, i.e., it is able to refresh the channel after changes in the topology of the network (e.g., as a consequence of mobility)—see Figure 1.

3.4 The distributed platform

For what concerns the distributed platform, `scafi` provides an object-oriented facade API that abstracts over the underlying actor-based architecture. The support of the actor model of computation as provided by Akka has been adopted as a starting point for the distributed middleware implementation. According to the operational semantics as developed in [8], an aggregate program can be turned into a single function to be repetitively executed in each device/actor, taking previous state, sensor values, and neighbours' states, and yielding a new state to be broadcasted to neighbours. Few lines of code are to be used in each device to configure, boot, and observe the resulting computational fields.

```
val settings = Settings(...) // platform settings
val platform = PlatformConfigurator.setupPlatform(settings)

val system = platform.newAggregateApplication {
  channel(isSource, isTarget)
}

val device = system.newDevice(
  id = Utils.newId(),
  neighbourhood = Utils.discoverNbrs())
device.addSensor("source", () => true)
device.addSensor("target", () => false)

(device.actorRef ? GetExport).onSuccess {
   case MsgExport(isInChannel) =>  // ...
}
```

Aggregate computing is a high-level model that can work with different architectures and be adopted in many distributed computing contexts: it can be used to build systems working in a peer-to-peer or server-based fashion, and also provides spatial abstractions to model situation and space-aware neighbouring relations.

Conclusions

Aggregate programming is a paradigm that can significantly contribute to the development of applications in a variety of domains which, besides, are not limited to spatially-situated systems. In addition to crowd engineering and related services for mass public events, other typical scenarios include: distributed coordination in auxiliary tactical networks supporting humanitarian interventions; self-recovery of interdependent enterprise services [5]; and complex building automation. In fact, aggregate programming may be a valuable tool whenever robust adaptivity, collective self-organisation, and independence to underlying architectural details are desirable. Aggregate programming is essentially a generalisation of previous approaches falling under the umbrella of so-called *macro-programming*, reviewed in [2].

Future works encompass deep investigation of foundational aspects related to static and dynamic properties, development of reusable APIs for effective and efficient engineering of complex applications, and creating a whole toolchain based on `scafi`, most notably, up to a scalable edge and cloud-based support—using techniques such as those in [11].

4. REFERENCES

[1] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.

[2] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16. IGI Global, 2013.

[3] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 2015.

[4] J. Beal and M. Viroli. Space–time programming. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

[5] S. S. Clark, J. Beal, and P. Pal. Distributed recovery for enterprise services. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 111–120, Sept 2015.

[6] F. Damiani and M. Viroli. Type-based self-stabilisation for computational fields. *Logical Methods in Computer Science*, 11(4):1–53, 2015.

[7] F. Damiani, M. Viroli, and J. Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117:17 – 44, 2016.

[8] 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.

[9] 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.

[10] J. Fruin. *Pedestrian and Planning Design*. Metropolitan Association of Urban Designers and Environmental Planners, 1971.

[11] C. Meiklejohn, S. H. Haeri, and P. V. Roy. Declarative, sliding window aggregations for computations at the edge. In *2016 13th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 32–37, Jan 2016.

[12] B. C. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *ACM Sigplan Notices*, volume 45, pages 341–360. ACM, 2010.

[13] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[14] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 2013.

[15] D. Pianini, M. Viroli, and J. Beal. Protelis: Practical aggregate programming. In *Proceedings of ACM SAC 2015*, pages 1846–1853, Salamanca, Spain, 2015. ACM.

[16] 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.

[17] M. Viroli and F. Damiani. A calculus of self-stabilising computational fields. In *Coordination Languages and Models*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag, June 2014.

[18] F. Zambonelli and M. Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. In *Autonomic communication*, pages 44–57. Springer, 2005.