Proceedings of the Federated Conference on Computer Science and Information Systems pp. 1495–1504
DOI: 10.15439/2016F407
ACSIS, Vol. 8. ISSN 2300-5963

Simulating Large-scale Aggregate MASs with Alchemist and Scala

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

Danilo Pianini
Università di Bologna, Italy
danilo.pianini@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 based on opportunistic IoT-based applications, promote aggregate programming, a development approach for distributed systems in which the collectivity of devices is directly targeted, 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). Most specifically, we argue that aggregate programming is particularly suitable for building models and simulations of complex large-scale reactive MASs. Accordingly, in this paper we describe SCAFI (Scala Fields), a Scala-based API and DSL for aggregate programming, and its integration with the ALCHEMIST simulator, and usage scenarios in the context of smart mobility.**
Keywords — aggregate programming, Scala, DSL, simulation.

I. INTRODUCTION

APPLYING multiagent systems (MASs) in the context of large-scale distributed systems is known to be hard in general, due to the ineluctable need to take into account issues such as communication, robustness, consistency and performance. The situation is then becoming harder and harder especially in recently emerging distributed computing scenarios, such as pervasive computing or IoT, 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 source of unpredictability. Achieving a sound development of MAS 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.

Recent research in collective adaptive software systems proposed *aggregate computing* [1] as a promising approach generalising over several prior models and languages addressing computations over collections of spatially-situated systems [2]. 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. Aggregate computing can be formally grounded in the field calculus [3], a core language able to express complex patterns of information diffusion and aggregation. Resiliency is guaranteed by self-organisation: aggregate programs can be compiled into repetitive local tasks to be executed by the single agent, promoting identification of robust building blocks of aggregate behaviour [4].

Though naturally applicable to swarm-like reactive MASs, aggregate computing is also of interest when stronger notions of agency enter the picture: as discussed in [5], aggregate programs can be seen as "aggregate plans," namely, operational instructions for deliberative agents that guide the cooperative behaviour of a team.

In order to more deeply investigate the impact of this new paradigm to the mainstream development of large-scale MASs, in this paper we explore and propose an extension of the Alchemist simulator [6] with `scafi`[1] [7], a Scala framework that provides an internal domain-specific language (DSL) for specifying aggregate computations via a simple API that well integrates with the advanced typing features of Scala (inference, genericity, implicits) and its library. This allows to smoothly simulate complex collective adaptive behaviours on top of Scala mainstream programming.

The remainder of this paper is organised as follows: Section 2 introduces aggregate programming, Section 3 presents the `scafi` framework and example specifications, Section 4 depicts Alchemist and the integration of `scafi`, Section 5 discusses case studies in the context of smart mobility, and Section 6 presents related works before finally drawing conclusions.

II. AGGREGATE PROGRAMMING

Aggregate programming [1] is a novel approach to (large-scale) distributed systems engineering 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 computation is carried on by the interaction of individual entities. It essentially solves the inverse, global-to-local mapping problem.

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

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. More specifically, as realised by space-time programming approaches [2], logical or physical neighbouring of nodes can be exploited to make interaction implicit. In addition, this notion is instrumental for the conceptual connection with systems where locality might play a major role in communication.

The main programming abstraction in aggregate programming is the *computational field* [3] (or field for short), a notion already used in the MAS community [8], [9]. Generalising the notion of (gravitational, electromagnetic) field in physics, a computational field is a function that, at a given moment in time, maps each point in space to a computational object; when considering the space discretised by a networked set of situated agents, each sample represents the outcome of computation for that agent. The key insight of the approach consists in the ability to specify collective behaviours (i.e., aggregate computations) by algorithms expressed as a functional composition of fields: from input fields representing information sensed from the environment, up to output fields representing some form of actuation. In other words, aggregate computations are represented by a declarative specification of functional operations involving collective data structures, though, under-the-hood, they are turned into repetitive, gossip-like interactions between individual agents—namely, relying on known low-level self-organisation patterns [10].

This approach is shown to support a solid engineering methodology, in which composable and reusable high-level library components of aggregate behaviour can be defined that are provably resilient [4].

III. Aggregate Programming in Scala

The aggregate computing idea can be naturally supported by a programming language, used to specify aggregate behaviours. Among different choices one can make to frame one such language (as a DSL, as an API, and so on), in this paper we explore the idea of using the Scala programming language [11] as the host language for building an aggregate programming platform. This is motivated by both technical and practical reasons.

Scala is a modern language for the JVM which integrates the object-oriented and functional paradigms in a seamless way, hence we can combine the typically rich and expressive OO libraries and data structures, with functional programming as promoted by aggregate computing. Scala has a powerful and expressive type system, combining the advantages of static type checking with a concise syntax for productivity—thanks to type inference, the implicits system, generic and functional programming features, and ad-hoc syntactic sugar. This allows library designers to create Scala APIs which actually have a DSL-like flavour, which are thus perceived by users as "embedded languages." Moreover, Scala is now becoming a standard de facto for the construction of platforms for distributed processing (frameworks like Akka actors, and Apache Kafka, Storm and Spark are essentially Scala-based): this makes it the ideal language to target a platform for aggregate computing—though this issue is not discussed here further.

Accordingly, we propose `scafi` (Scala fields) [7], a framework consisting of two main parts:

1) *aggregate programming support*, by a Scala-internal DSL that provides a syntax and the corresponding semantics for the constructs of the computational field calculus [3], by which aggregate computations are naturally expressed and seamlessly combined in code; and
2) *aggregate platform support*, allowing configuration and execution of such code in actual distributed setups.

`scafi` explicitly addresses the construction of concrete aggregate computing applications: however, it can also play the role of a language to validate complex MAS algorithms, as meta-model for simulations—in the next section, in fact, we shall describe its integration with a full featured simulator, Alchemist.

A. Computational field calculus in Scala

An aggregate system consists of a (possibly mega-scale) number of computational devices or agents, all executing the same aggregate program at asynchronous rounds of computation. According to contextual information (e.g, sensor values), different such computational devices may take different branches of computation, i.e., computing sub-fields in different domains of execution. Interaction depends on a notion of locality, i.e., a device can 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 to the neighbourhood: the shape of this state, and how it affects and gets affected by computations, is precisely defined by our language semantics [3]. The basic primitives of the field calculus (described below, in turn) are declared in the `Constructs` trait, implemented by the framework and mixed-in in any library of user-defined aggregate functions:

```
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 constructs can be understood and described according to two complementary viewpoints:

1) *Local viewpoint* – refers to the traditional device-centric interpretation where an aggregate computation is considered in the context of a single device. The operational semantics of the field calculus is implemented according to this stance.
2) *Global viewpoint* – it corresponds to the natural semantics and refers to the aggregate-level interpretation of programs as computations running on whole fields

(i.e., spatial data structures mapping each device to some computational object).

B. Working with basic constructs

The most trivial program 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 evaluating that value everywhere; concretely, it results in the string `"Hello, World"` being the return value of the local computation of every device in the system.

Construct `sense` provides the means for reading a value from a local sensor, which enables context-sensitive behaviours. Expression

```
sense[Double]("temperature")
```

gets in any device a double value from the temperature sensor, creating the field of temperatures.

Change over time of a field can be realised via the `rep` construct, which produces a dynamically evolving field by repeatedly applying a state-transformation function. For example, it could be used for counting how many rounds a device performed 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 the frequency at which devices compute rounds and hence send messages to neighbours can vary over time and from agent to agent: the aggregate computing model generally assumes partial synchronicity [12], though in most cases even full asynchrony of rounds can be assumed.

Communication with the neighbourhood is achieved via `nbr`, which gives a map from neighbouring devices to their value of the argument—essentially an observation primitive. Construct `nbr` has to be nested inside a `foldhood` operation, which reduces one such map back to a single value via a monoidal reduction (on top of it, derived `minHood`, `sumHood` and others are defined and will be used in next sections). Example applications of `foldHood` are as follows:

```
// 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") } }
```

In addition to local sensors, there is a notion of "environmental" sensor. `nbrvar` allows to extract values from a neighbouring sensor, which gives a sample for each neighbour. Thus, similarly to `nbr`, `nbrvar` has to be used within a `foldhood` operation.

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

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

Operation `branch` splits the spatial domain of devices into two parts, or rather two sub-teams, according to a boolean field expressing some condition. Each of the two parts, compute a different sub-field in complete isolation. For example, if we would like to execute some aggregate computation only on a subset of the devices of the network (the complementary subset must not participate), we need a partition:

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

Construct `aggregate` is used to define the body of a new function that should work on whole fields. The devices running a given aggregate function constitute a partition, i.e., they are able to interact with each other via `nbr`. As an example, `branch` could be entirely rewritten using `aggregate`:

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

The symbol `mux` used in this example is a built-in operator that provides a purely functional multiplexer.

C. Working with combinations

More elaborate aggregate behaviours can be defined by compositions of the basic constructs. In addition, specific parts of the program logic can be encapsulated into Scala functions. Simple examples involving these features include counting neighbours except the device itself:

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

sumHood{ mux(isMe){0}{1} }
```

The paradigmatic example of computational field is known as gradient [10], computing in each node the distance (hop-by-hop, or estimated) from the nearest node where a `source` field holds a true value:

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

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

D. Scaling with complexity

Even though the basic constructs of the field calculus are somewhat low-level, they can be further composed so as to define reusable library components that provide higher-level behaviours—in fact, the functional character of the approach promotes systematic factorisation of behaviour into reusable layers of increasing abstraction.

An initial set of general coordination operators has been identified in [1], [4]. These operators capture common patterns of distributed computation and also enjoy the *self-stabilisation* property, which ensures that a system, independently of the current state, will eventually reach a stable state in finite time

that is not affected by transitory events. Moreover, as the self-stabilisation property is preserved by composition [13], it is formally guaranteed that also composite structures and algorithms self-stabilise.

On top of such resilient building blocks, a sound development API can be defined, which in turn can be used to implement application-specific aggregate behaviours.

1) Gradient-cast: `G` simultaneously performs two tasks: i) builds a distance-gradient from the source (`src`) according to `metric`, and ii) builds accumulated values via `acc` along the gradient starting from `init` at the `src`. In `scafi`, it can be encoded as follows:

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

The generic type `V` (and, in this case, also `Tuple2[+A,+B]`) must have an (implicit or explicit) instance of the `OrderingFoldable` type-class available so that `minHoodPlus` can work out the minimum value on the neighbourhood (by convention, `*hoodPlus` operators exclude the device itself from the neighbours set).

A `broadcast` operation can easily get built on top of `G`:

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

In the following example, we leverage `broadcast`, to diffuse across the whole network the distance between two devices:

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

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

def isSource = sense[Boolean]("source")
def isObstacle = sense[Boolean]("obstacle")

distBetween(isSource, isObstacle)
```

and, most notably, we could realise an algorithm that builds a `width`-wide channel connecting a source and a destination:

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

2) Converge-cast: `C` is the dual of `G`: it collects information distributed across space by `acc`umulating values down a `potential` field, starting with `local` at the sources (i.e., devices with no parent, located at the edge of the `potential` field):

```
def C[V](potential: V, acc: (V,V)=>V, local: V, Null: V)
        (implicit ev: OrderingFoldable[V]): V = {
  rep(local){ v =>
    acc(local, foldhood(Null)(acc){
      mux(nbr(findParent(potential)) == mid()){
        nbr(v)
      } {
        nbr(Null)
      }
    })
  }
}

def findParent[V](potential: V)
                 (implicit ev: OrderingFoldable[V]): ID = {
  mux(ev.compare(minHood{ nbr(potential) }, potential)<0 ){
    minHood{ nbr{ Tuple2[V,ID](potential, mid()) } }._2
  }{
    Int.MaxValue
  }
}
```

`C` and `G` can be combined to originate a self-stabilising `summarise` operator that first collects information across the space and then propagates back the computed summary:

```
def summarize(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 =
  summarize(sink, (a,b)=>{a+b}, value, 0.0) /
    summarize(sink, (a,b)=>a+b, 1, 0.0)
```

3) Time-decay: The `T` operator can be used to condense information across time by decreasing the `initial` field according to a `decay` function:

```
def T[V](initial: V, floor: V, decay: V=>V)
        (implicit ev: Numeric[V]): V = {
  rep(initial){ v =>
    ev.min(initial, ev.max(floor, decay(v)))
  }
}

def T[V](initial: V)
        (implicit ev: Numeric[V]): V = {
  T(initial, ev.zero, (t:V)=>ev.minus(t, ev.one))
}
```

Given such a function, the implementation of a `timer` is direct. With `timer`, a `limitedMemory` function can be defined, computing `value` until `timeout` has expired and `expValue` thereafter, effectively realising a memory limited in time.

```
def timer[V](length: V)
            (implicit ev: Numeric[V]) = T[V](length)

def limitedMemory[V,T](value: V, expValue: V, timeout: T)
                      (implicit ev: Numeric[T]) = {
  val t = timer[T](timeout)
  (mux(ev.gt(t, ev.zero)){value}{expValue}, t)
}
```

4) Sparse-choice: `S` can be used to create partitions and for selecting sparse subsets of devices in space. Essentially, it realises a local leader election, where `grain` is the mean distance between two leaders and `metric` represents the notion of distance.

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

The implementation uses `randomUid` to generate a field of unique identifiers:

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

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 between devices for leadership:

```
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}}
      }
    }
  }
}
```

5) Restriction in space: We have already encountered the operator for doing domain restriction, which is `branch`. With it, a number of interesting computations can be achieved:

```
// Compute distance from 'src', avoiding obstacles
def distAvoidObstacles(src: Boolean, obs: Boolean): Double =
  branch(obs){ Double.PositiveInfinity }{ distanceTo(src) }

// Perform a broadcast within a particular 'region'
def bcastRegion[V](region: Boolean, src: Boolean, v: V)
                  (implicit ev: OrderingFoldable[V]): Option[V] =
  branch[Option[V]](region){
    Some[V](broadcast(src, v))
  }{ None }

// Measure the size of connected components of a region
def groupSize(region: Boolean): Double =
  branch(region){ summarize(S(1,0),_+_,1,0) }{ Double.NaN }

// Remember whether an event has recently occurred
def recentEvent(event: Boolean, timeout: Int): Boolean =
  branch(event){ true } { timer(timeout)>0 }
```

IV. Alchemist as simulation platform

Testing, debugging, and performance assessment prior to actual deployment are key components of a good software engineering process, and aggregate programming makes no exception. However, since the primary target for this emerging paradigm are distributed and situated systems, conventional testing and debugging tools are hardly enough, in particular falling short at capturing the interaction among devices and between them and the underlying environment. In this situation, simulation emerges as a valuable tool for all the phases of software development: early testing and debugging, integration testing, and performance assessment. Of course, simulation cannot entirely capture the complexity of the real system (much like the classic unit testing cannot test every possible situation in classic application development), nevertheless the desiderata is to be as close as possible to a real situation. There are two relevant dimensions in this regard: first, the simulated environment must capture the most relevant aspect of the distributed system under design; second, the code that the simulator executes must resemble as closely as possible the production code. Considering both dimensions, we picked Alchemist [6].

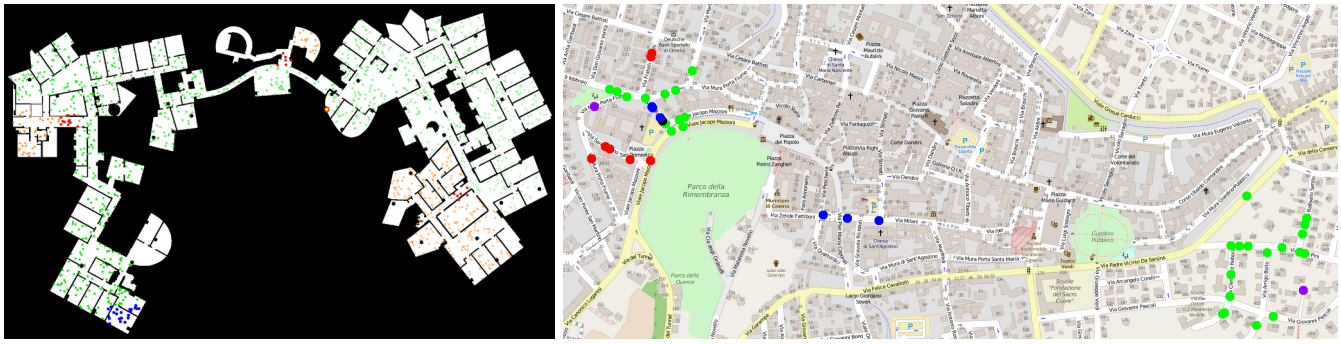
Alchemist is an event-driven simulator, mostly written in Java, tailored to the simulation of pervasive systems with a focus on performance. The model, albeit originally inspired by chemistry, supports complex environments, several flavors of node mobility, and advanced network models. Particularly interesting for real world applications is the possibility of exploiting map data from OpenStreetMap, navigating nodes along existing GPS traces as well as along roads (distinguishing among a handful of vehicles types). Also useful is the support for converting indoor images to Alchemist environments with physical obstacles. Figure 1 shows typical instances of environments frequently simulated with Alchemist.

The Alchemist computational model is generic (it is rather a meta-model), and requires a so called "incarnation" to be developed in order to actually execute simulations. An incarnation is a mapping between the Alchemist meta-model concepts and the concrete entities that the user is interested in simulating. Alchemist, at the time of writing, ships two incarnations, one of them tailored to aggregate programming supporting the simulation of (possibly mobile) Protelis [14] programmed devices.

A. Interfacing Alchemist and Scala

Our goal in interfacing `scafi` and Alchemist was to be able to feed it with the production Scala code, injecting it directly into the simulated environment. A few factors made such integration quite straightforward:

1) Scala and Java are both hosted in the JVM and feature full, bidirectional interoperability;
2) the `scafi` architecture neatly separates its core from the actor-based network backend, this design was key in our pursue to sharing interpreter and Scala code between the actor platform and the simulator;



(a) (b)

Fig. 1: Typical multi-agent scenarios supported by Alchemist: a very dense and intricate indoor environment (Figure 1a), and an urban environment (Figure 1b).

Alchemist meta-model	**Protelis incarnation**	**`scafi` incarnation**
Environment: container of nodes and network model	-	-
Network model	-	-
Node: container of reactions and molecules	Device: container of events and environment variables	Device: container of events and environment variables
Reaction: set of conditions that, if matched, triggers a set of actions with some time distribution	Event	Event
Condition	-	-
Action: any change of the environment	Any Alchemist action, or the execution of a computation round of a Protelis virtual machine, or the dispatch of results to neighbours	Any Alchemist action, or the execution of a `scafi` computation round, included message dispatching
Time distribution	-	-
Molecule, associated with a concentration	Environment variable, associated with a value	Environment variable, associated with a value
Concentration	Value: any Java object	Value: any Scala object

Fig. 2: Mapping between the Alchemist meta-model entities and the Protelis and `scafi` incarnation concepts. Omitted cells indicate that the Alchemist concept is inherited as-is, with no change. Similarities between the incarnations hosting the two aggregate languages are immediately clear.

3) the Protelis incarnation in Alchemist provided an architectural template, that led to a quick identification of the proper conceptual mapping to operate when building the incarnation.

Figure 2 summarizes the mapping effort between Alchemist entities and `scafi` ones, also comparing with the existing Protelis incarnation. As the reader may expect, several entities share the same meaning between the two aggregate programming languages, and as such we believe that an incarnation skeleton for any aggregate programming language could be crafted in Alchemist, further easing the integration of any JVM-hosted aggregate programming language interpreter. In particular, note that sensing of information is supported by means of environment variables in Alchemist, whose evolution over time is controlled when configuring an Alchemist simulation, and which binds to the behaviour of construct `sense` in `scafi`.

V. Case Study

Our goal in this work is to demonstrate the feasibility of using `scafi` in conjunction with Alchemist to realise complex simulations. As such, we do not aim at presenting novel scenarios, but rather we do explain how to recreate some of the results available in literature, validating the proposed framework and showing how convenient is to express complex collective adaptive systems with aggregate computing, `scafi`, and the available APIs. In particular, we focus on two examples that have already been implemented in Protelis and simulated in Alchemist:

1) an example application of smart vehicle counting, which showcases the usage of higher order functions and the possibility of tuning the computational round execution

frequency with Alchemist. The proposed code is derived from [15];
2) an urban crowd tracking scenario featuring the combined usage of all fundamental self-stabilising building blocks exposed in Section III-D [1].

The goal of these simulations, performed prior to actual deployment, would be to evaluate whether the propose collective adaptive system would provide an effective and efficient service for the application at hand.

A. Vehicle counting

Consider a highway, where a sensor is deployed to monitor the number of vehicles passing nearby. We suppose that the vehicles are equipped with electronic components that make them connected with all the other compatible devices within a certain range. We do not consider connectivity issues (which are beyond the scope of this work), and make instead the assumption that every device (vehicle or sensor) is able to communicate with every neighbour within a certain distance. We suppose that all devices are running the following `scafi` "virtual machine", which is able to import any injected procedure from the `snsInjectedFun` sensor:

```
// Dynamically import any computation
def snsInjectedFun: ()=>Double = sense("injectedFun")
// True where the data should get aggregated
def snsInjectionPoint: Boolean = sense("injectionPoint")
// 1 for patrons, 0 otherwise
def snsShouldCount: Double = mux(sense("patron")){1}{0}
// Detection range in meters
def snsRange: Double = 30

def virtualMachine(): Double = {
  deploy(snsRange,snsInjectionPoint,snsInjectedFun,()=>0.0)
}

def deploy[T](range:Double, source:Boolean,
              g: ()=>T, noOp: ()=>T)
             (implicit ev: OrderingFoldable[T]): T = {
  val f: ()=>T = branch(distanceTo(source) < range) {
    G(source, g, identity[()=>T], nbrRange())
  }{ noOp }
  f()
}
```

To monitor the number of nearby vehicles, the sensor device injects the following function, which relies on the `C` building block to realise a convergent sum:

```
def countPatrons() = {
  C[Double](potential = distanceTo(snsInjectionPoint),
      acc = _+_, local = snsShouldCount, Null = 0.0)
}
```

In this scenario, the sensor injects such function after two seconds of simulations, due to a congested block of traffic coming, and turns it off after 10 seconds (by injecting a function returning 0). We executed the same experiment multiple times, varying the round frequency. Figure 3 summarises the results. As expected, values get much closer to reality when the sampling frequency is very high. Moreover, the algorithm exploited to implement `C` is based on a spanning tree, that by its nature is sensible to changes in the network topology: these

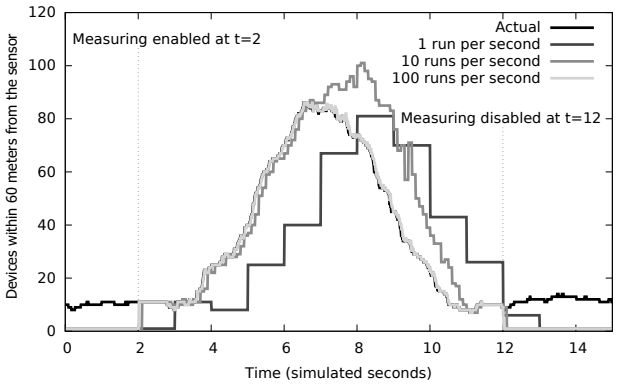


Fig. 3: Number of vehicles counted using the convergent, distributed sum based on `C`. As expected, higher frequencies lead to more precise measurements.

are responsible of the peak in the device count observed with devices running at 1Hz.

B. Crowd tracking in an urban scenario

As second example, consider an IoT environment that provides services for crowd safety at a mass public event, such as a marathon. Such events pose challenging safety issues, because the movement of people in crowded and constrained environments often creates emergent zones of dangerous overcrowding where any small incident can create a panic or stampede that injures or kills people. We simulate here a possible crowd safety service running in an IoT urban environment.

We define three possible crowding levels;

```
val (high,low,none) = (2,1,0) // crowd level
```

a function locally estimating crowd density;

```
def unionHoodPlus[A](expr: => A): List[A] =
  foldhoodPlus(List[A]())(_++_){ List[A](expr) }

def densityEst(p: Double, range: Double): Double = {
  val nearby = unionHoodPlus(
    mux (nbrRange < range) { nbr(List(mid())) } { List() }
  )
  nearby.size / p / (Math.PI * Math.pow(range,2))
}
```

a function mapping each local area to a danger level, depending on the average density sensed locally;

```
def managementRegions(grain: Double,
     metric: => Double): Boolean = S(gran,metric)

def dangerousDensity(p: Double, r: Double) = {
  val mr = managementRegions(r*2, () => { nbrRange })
  val danger = average(mr, densityEst(p, r)) > 2.17 &&
    summarize(mr, (_:Double)+(_:Double), 1 / p, 0) > 300
  mux(danger){ high }{ low }
}
```

and a function yielding true if a situation of danger has remained active for enough time.

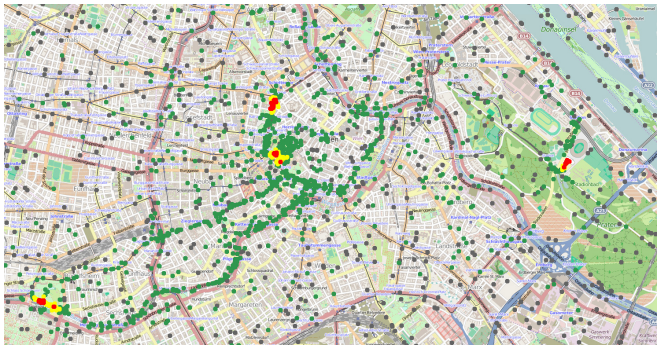


Fig. 4: A snapshot, taken from [1], of Alchemist executing the crowd warning application. Grey dots are stationary devices, not participating the system. Green dots are users in safe areas, red dots users in dangerous areas, and yellow dots are users who are being warned.

```
def recentTrue(state: Boolean, memTime: Double): Boolean = {
  branch(state) {
    true
  }{
    limitedMemory[Boolean,Double](started, false, memTime)._1
  }
}

def crowdTracking(p: Double, r: Double, t: Double) = {
  val crowdRgn = recentTrue(densityEst(p, r)>1.08, t)
  branch(crowdRgn){ dangerousDensity(p, r) }{ none }
}
```

With all those ingredients, we warn the users of those devices located near areas which have remained crowded for a long time.

```
def crowdWarning(p: Double, r: Double,
                 warn: Double, t: Double): Boolean = {
  distanceTo(crowdTracking(p,r,t) == high) < warn
}
```

For the sake of simplicity, the numbers we used for the estimates had been directly written in code. They could get extracted and substituted by parameters, the values proposed are derived from literature [16]. The actual simulation is composed of 1000 stationary devices embedded into the environment plus 1479 mobile personal devices, each following a smartphone position trace collected at the 2013 Vienna marathon [17], [18]. Figure 4 shows a sample screenshot of the system deployed.

VI. Related Work

A. Computational fields and aggregate programming

A wide range of existing approaches to aggregate programming have been proposed, including such diverse approaches as abstract graph processing (e.g., [19]), declarative logic (e.g., [20]), map-reduce (e.g., [21]), streaming databases (e.g., [22]), and knowledge-based ensembles (e.g., [23])—for a detailed review, see [24]. Most of them, however, have been too specialized for particular assumptions or applications to be able to address the aggregate programming challenge at its full complexity in a wide range of different environments.

A unifying model based on *computational fields* has been identified as a generalization of a wide range of existing approaches, surveyed in [24]. Formalized as the computational field calculus [3], this universal language provides a theoretical foundation on which real aggregate programming platforms can be built: both Protelis [14] and `scafi` are practical instances of such calculus.

B. Aggregate programming and multi-agent systems

Aggregate programming targets collective behaviour of systems, which in the MAS literature have typically been addressed in various ways. On the one hand, we have approaches facing the design of relatively small systems of deliberative/cognitive agents: coordination mechanisms and tools (e.g. via artifacts [25] or protocols [26]), social/organisational norms [27], commitments [28], and so on. They provide declarative constraints to agent interaction (abstracting away from the step-by-step operational behaviour of the aggregate), and strongly rely or deal with autonomy of agents, assuming agents have inner mechanisms to dynamically adapt their behaviour to the specific contingency, up to the point of deviating from a previously agreed cooperative behaviour.

On the other hand, collective behaviour of large-scale MASs is mostly studied in terms of swarm-intelligence techniques, assuming that agents are reactive and perform repetitive tasks, with the problem of designing local agent behaviours that can end up in the desired global tasks [29].

Aggregate programming somewhat sits in between these two apparently unreconcilable views of a self-adaptive MASs: it aims at devising a methodology by which the collective behaviour of large-scale ensembles of autonomous, deliberative agents, can be designed so as to manifest inherent properties of self-adaptation, resiliency, and openness.

C. Simulation of aggregates

We claimed in Section IV that simulation is a key step in the development of aggregate programming applications. There are many kinds of simulation tools available: they either provide programming/specification languages devoted to ease construction of the simulation process, especially targeting computing and social simulation (e.g. as in the case of multi-agent based simulation [30], [31], [32], [33], [34]), or they stick to quite foundational computing languages to better tackle performance, mostly used in biology-oriented applications [35], [36], [37], [38].

Alchemist is a discrete-event simulator (DES), since it combines a continuous time base with the description of system dynamics by distinguished state changes [39]. The class of DES more related to our approach are those commonly used to simulate biological-like systems, by which in fact Alchemist was originally inspired. A recent overview of them is available in [38], which takes into account: DEVS [40], Petri Nets [36], State Charts [41], and stochastic π-calculus [35].

VII. Conclusions and future works

In this paper, we introduced `scafi`, a Scala based API and DSL for aggregate programming, equipped with an actor based platform and integrated with the Alchemist simulator. We described the main features of the API/DSL, and demonstrated how `scafi` can be used to realise reusable building blocks that ease the creation of aggregate programs. We presented the integration between `scafi` and Alchemist, a discrete-event simulator targeting pervasive systems, detailing the mapping between the Alchemist meta-model and the concrete `scafi` entities. We were able to push the integration to the point that there exists no difference between the production code and the code required to perform a simulation. We argue that such a deep level of integration will improve the engineering practices when it comes to leveraging aggregate programming for building actual systems, allowing for debugging and testing on a centralized testing platform prior to deployment. We validated the approach by translating complex examples found in literature in `scafi`, using Alchemist as simulation platform.

Further development of this research includes a refinement of the current `scafi` architecture and of its simulator integration. While Alchemist is already publicly available, `scafi` is set to be ready for the general public shortly, as well as the Alchemist incarnation supporting the execution of `scafi` programs in a controlled environment. We expect `scafi` and the related tool chain to boost the adoption of aggregate programming as a new paradigm for the engineering of complex, pervasive systems, such as the typical Internet of Things applications.

References

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

[2] J. Beal and M. Viroli, "Space–time programming," *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 373, no. 2046, 2015. doi: 10.1098/rsta.2014.0220. [Online]. Available: http://rsta.royalsocietypublishing.org/content/373/2046/20140220

[3] F. Damiani, M. Viroli, and J. Beal, "A type-sound calculus of computational fields," *Science of Computer Programming*, vol. 117, pp. 17 – 44, 2016. doi: http://dx.doi.org/10.1016/j.scico.2015.11.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642315003573

[4] 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*. IEEE, Sept 2015. doi: 10.1109/SASO.2015.16 pp. 81–90.

[5] M. Viroli, D. Pianini, A. Ricci, P. Brunetti, and A. Croatti, "Multi-agent systems meet aggregate programming: Towards a notion of aggregate plan," in *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, ser. Lecture Notes in Computer Science, Q. Chen, P. Torroni, S. Villata, J. Hsu, and A. Omicini, Eds. Springer International Publishing, 2015, vol. 9387, pp. 49–64. ISBN 978-3-319-25523-1. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-25524-8_4

[6] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with Alchemist," *Journal of Simulation*, 2013. doi: 10.1057/jos.2012.27. [Online]. Available: http://www.palgrave-journals.com/jos/journal/vaop/full/jos201227a.html

[7] R. Casadei and M. Viroli, "Towards aggregate programming in scala," in *First Workshop on Programming Models and Languages for Distributed Computing*. ACM, 2016, p. 5.

[8] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The TOTA approach," *ACM Trans. on Software Engineering Methodologies*, vol. 18, no. 4, pp. 1–56, 2009. doi: http://doi.acm.org/10.1145/1538942.1538945

[9] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson, "Pervasive ecosystems: a coordination model based on semantic chemistry," in *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, S. Ossowski, P. Lecca, C.-C. Hung, and J. Hong, Eds. Riva del Garda, TN, Italy: ACM, 26-30 March 2012. ISBN 978-1-4503-0857-1 pp. 295–302.

[10] 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*, vol. 12, no. 1, pp. 43–67, 2013. doi: 10.1007/s11047-012-9324-y

[11] 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," Tech. Rep., 2004.

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

[13] M. Viroli and F. Damiani, "A calculus of self-stabilising computational fields," in *Coordination Languages and Models*, ser. LNCS, eva Kühn and R. Pugliese, Eds. Springer-Verlag, Jun. 2014, vol. 8459, pp. 163–178, proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June. Best Paper of Discotec 2014 Federated conference.

[14] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *Proceedings of ACM SAC 2015*. Salamanca, Spain: ACM, 2015, pp. 1846–1853.

[15] F. Damiani, M. Viroli, D. Pianini, and J. Beal, "Code mobility meets self-organisation: A higher-order calculus of computational fields," ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 9039, pp. 113–128. ISBN 978-3-319-19194-2. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19195-9_8

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

[17] B. Anzengruber, D. Pianini, J. Nieminen, and A. Ferscha, "Predicting social density in mass events to prevent crowd disasters," in *Social Informatics*, ser. Lecture Notes in Computer Science, A. Jatowt, E.-P. Lim, Y. Ding, A. Miura, T. Tezuka, G. Dias, K. Tanaka, A. Flanagin, and B. Dai, Eds. Springer International Publishing, 2013, vol. 8238, pp. 206–215. ISBN 978-3-319-03259-7. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03260-3_18

[18] D. Pianini, M. Viroli, F. Zambonelli, and A. Ferscha, "HPC from a self-organisation perspective: The case of crowd steering at the urban scale," in *High Performance Computing Simulation (HPCS), 2014 International Conference on*, July 2014. doi: 10.1109/HPCSim.2014.6903721 pp. 460–467.

[19] R. Gummadi, O. Gnawali, and R. Govindan, "Macro-programming wireless sensor networks using kairos." in *Distributed Computing in Sensor Systems (DCOSS)*, 2005, pp. 126–140.

[20] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *IEEE International Conference on Intelligent Robots and Systems (IROS '07)*, 2007, pp. 2794–2800.

[21] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[22] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *Workshop on Mobile Computing and Systems Applications*, 2002.

[23] R. D. Nicola, M. Loreti, R. Pugliese, and F. Tiezzi, "A formal approach to autonomic systems programming: The SCEL language," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 2, pp. 7:1–7:29, Jul. 2014. doi: 10.1145/2619998. [Online]. Available: http://doi.acm.org/10.1145/2619998

[24] 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*, M. Mernik, Ed. IGI Global, 2013, ch. 16, pp. 436–501. ISBN 978-1-4666-2092-6 A longer version available at: `http://arxiv.org/abs/1202.5509`.

[25] M. Viroli, A. Omicini, and A. Ricci, "Engineering MAS environment with artifacts," in *2nd International Workshop "Environments for Multi-*

Agent Systems" (E4MAS 2005), D. Weyns, H. V. D. Parunak, and F. Michel, Eds., AAMAS 2005, Utrecht, The Netherlands, 26 Jul. 2005.

[26] A. K. Kalia and M. P. Singh, "Muon: designing multiagent communication protocols from interaction scenarios," *Autonomous Agents and Multi-Agent Systems*, vol. 29, no. 4, pp. 621–657, 2015. doi: 10.1007/s10458-014-9264-2. [Online]. Available: http://dx.doi.org/10.1007/s10458-014-9264-2

[27] A. Artikis, M. J. Sergot, and J. V. Pitt, "Specifying norm-governed computational societies," *ACM Trans. Comput. Log.*, vol. 10, no. 1, 2009. doi: 10.1145/1459010.1459011. [Online]. Available: http://doi.acm.org/10.1145/1459010.1459011

[28] A. U. Mallya and M. P. Singh, "An algebra for commitment protocols," *Autonomous Agents and Multi-Agent Systems*, vol. 14, no. 2, pp. 143–163, 2007. doi: 10.1007/s10458-006-7232-1. [Online]. Available: http://dx.doi.org/10.1007/s10458-006-7232-1

[29] H. V. D. Parunak, S. Brueckner, R. S. Matthews, and J. A. Sauter, "Pheromone learning for self-organizing agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, vol. 35, no. 3, pp. 316–326, 2005. doi: 10.1109/TSMCA.2005.846408. [Online]. Available: http://dx.doi.org/10.1109/TSMCA.2005.846408

[30] S. Bandini, S. Manzoni, and G. Vizzari, "Agent based modeling and simulation: An informatics perspective," *Journal of Artificial Societies and Social Simulation*, vol. 12, p. 4, 2009. [Online]. Available: http://EconPapers.repec.org/RePEc:jas:jasssj:2009-69-1

[31] M. Schumacher, L. Grangier, and R. Jurca, "Governing environments for agent-based traffic simulations," in *Proceedings of the 5th international Central and Eastern European conference on Multi-Agent Systems and Applications V*, ser. CEEMAS '07. Berlin, Heidelberg: Springer-Verlag, 2007. doi: 10.1007/978-3-540-75254-7_17. ISBN 978-3-540-75253-0 pp. 163–172. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75254-7_17

[32] S. Bandini, S. Manzoni, and G. Vizzari, "Crowd Behavior Modeling: From Cellular Automata to Multi-Agent Systems," in *Multi-Agent Systems: Simulation and Applications*, ser. Computational Analysis, Synthesis, and Design of Dynamic Systems, A. M. Uhrmacher and D. Weyns, Eds. CRC Press, Jun. 2009, ch. 13, pp. 389–418. ISBN 978-1-4200-7023-1. [Online]. Available: http://crcpress.com/product/isbn/9781420070231

[33] M. J. North, T. R. Howe, N. T. Collier, and J. R. Vos, "A declarative model assembly infrastructure for verification and validation," in *Advancing Social Simulation: The First World Congress*, S. Takahashi, D. Sallach, and J. Rouchier, Eds. Springer Japan, 2007, pp. 129–140.

[34] E. Sklar, "Netlogo, a multi-agent simulation environment," *Artificial life*, vol. 13, no. 3, pp. 303–311, 2007.

[35] C. Priami, "Stochastic pi-calculus," *The Computer Journal*, vol. 38, no. 7, pp. 578–589, 1995.

[36] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989. doi: 10.1109/5.24143. [Online]. Available: http://dx.doi.org/10.1109/5.24143

[37] A. M. Uhrmacher and C. Priami, "Discrete event systems specification in systems biology - a discussion of stochastic pi calculus and devs," in *Proceedings of the 37th conference on Winter simulation*, ser. WSC '05. Winter Simulation Conference, 2005. ISBN 0-7803-9519-0 pp. 317–326. [Online]. Available: http://dl.acm.org/citation.cfm?id=1162708.1162767

[38] R. Ewald, C. Maus, A. Rolfs, and A. M. Uhrmacher, "Discrete event modeling and simulation in systems biology," *Journal of Simulation*, vol. 1, no. 2, pp. 81–96, 2007. [Online]. Available: http://dx.doi.org/10.1057/palgrave.jos.4250018

[39] B. P. Zeigler, *Theory of Modeling and Simulation*. John Wiley, 1976.

[40] B. Zeigler, *Multifaceted modelling and Discrete Event Simulation*. Academic Press, 1984.

[41] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987. doi: 10.1016/0167-6423(87)90035-9. [Online]. Available: http://dx.doi.org/10.1016/0167-6423(87)90035-9