

Simulating Large-scale Aggregate MASs with Alchemist and Scala

Roberto Casadei, Danilo Pianini, Mirko Viroli

`roberto.casadei12@studio.unibo.it`

`{danilo.pianini, mirko.viroli}@unibo.it`

ALMA MATER STUDIORUM—Università di Bologna

10th International Workshop on Multi-Agent Systems and Simulation (MAS&S'16)
September 14th, 2016 - Gdansk, Poland



- 1 Context and Motivation
- 2 Aggregate Programming
 - Basics
 - In Scala
- 3 Alchemist
- 4 Example
- 5 Conclusion



Outline

- 1 Context and Motivation
- 2 Aggregate Programming
 - Basics
 - In Scala
- 3 Alchemist
- 4 Example
- 5 Conclusion



Context: large scale situated systems



Motivation

Questions

- What would be the single agent goal / plan / program?
- How can we link agent behaviour with overall global effect?
- Can we reach predictable levels of resiliency?
- Which methods can support reuse and compositional behaviours?

A proper computational model

- Hides complexity under the hood
- Allows for describing the problem, not “hacking a solution”

A paradigm shift

Provide means for the designer to reason in terms of aggregate, rather than single agents

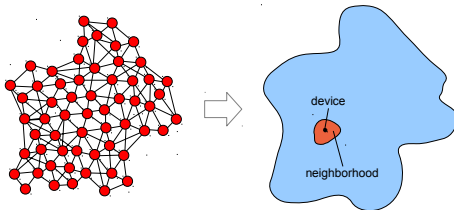
Outline

- 1 Context and Motivation
- 2 Aggregate Programming
 - Basics
 - In Scala
- 3 Alchemist
- 4 Example
- 5 Conclusion



Aggregate programming

Atomic manipulations of **computational fields**: a map $\text{Space} \Rightarrow \text{Value}$



- Based on programming languages rather than frameworks
- Complexity is offloaded to compilers and interpreters
- Compositional by design
- Based on Field Calculus (functional with higher order)
 - Computational-round based model
 - Clear operational semantics
 - Mathematical proofs about soundness, consistency, type systems...
- Novel abstractions, low level mechanisms $\Rightarrow$ steep learning curve



Scala

An attempt to address shortcomings and criticisms of Java

- General purpose, multi-paradigm language
- Very strong static typing
- Compiles in JVM bytecode
- Java interoperable
- Nice support to DSL definition

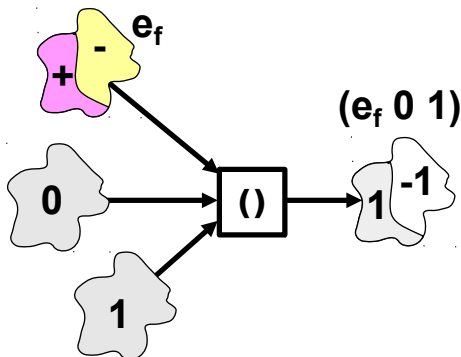


Basic constructs

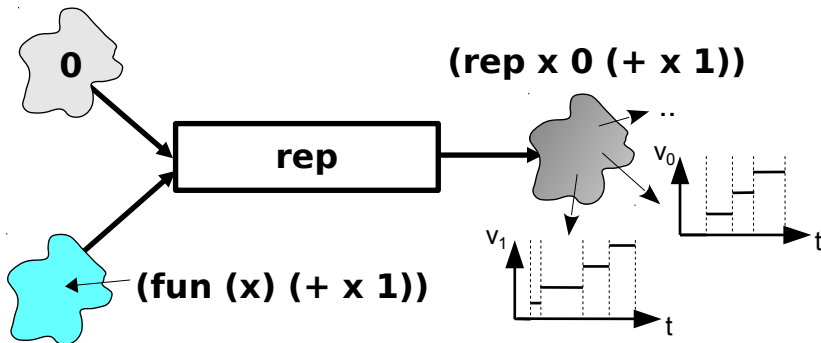
```
trait Constructs {  
  // Evolution over time (maintain state over rounds)  
  def rep[A](init: A)(fun: (A) => A): A  
  
  // Share and gather information from neighborhood  
  def nbr[A](expr: => A): A  
  def foldhood[A](init: => A)(acc: (A,A)=>A)(expr: => A): A  
  
  // Domain restriction  
  def branch[A](cond: => Boolean)(th: => A)(el: => A): A  
  
  // Access local and neighborhood sensors  
  def sense[A](name: LSNS): A  
  def nbrvar[A](name: NSNS): A  
}
```



How it works: apply functions over fields

$$e(0, 1)$$


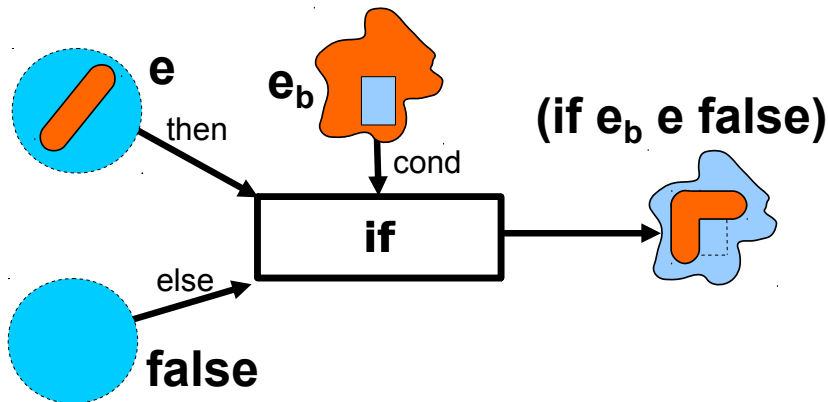
How it works: evolution over time

$$\text{rep}(0) \quad \{ \quad x \Rightarrow x + 1 \quad \}$$


$$\text{nbr}\{e\}$$


How it works: domain restriction (distributed `if`)

```
branch ( e ) { channel ( ) } { false }
```



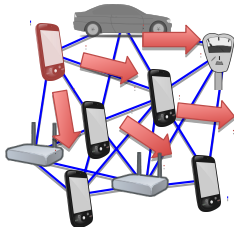
The channel computes in a different domain, but does not get disrupted!



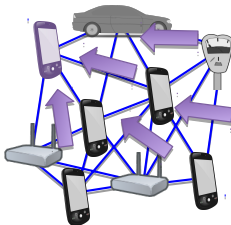
Get to the higher level

- These are the very low level mechanisms
- Upon them, three “building blocks” are defined
- They can be leveraged to realise reusable, higher level algorithms

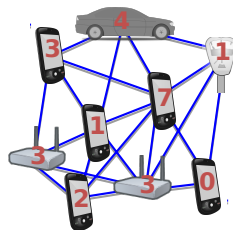
G: spreading



C: collecting



T: time decay



Examples with G

// A field of the same value

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

// A field of distances from an area (linear gradient in space)

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

// A field of the shortest distance between two areas

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



Examples with G and C

// A field of the sum of a value across the whole network

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

// A field of the average of a value across the whole network

```
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)
```



Examples with T

```
// A field of the time left to a certain moment
def timer[V] (length: V)
    (implicit ev: Numeric[V]) = T[V] (length)

// A field holding a value until the timer goes, and another
// value thereafter
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)
}
```



Outline

- 1 Context and Motivation
- 2 Aggregate Programming
 - Basics
 - In Scala
- 3 Alchemist**
- 4 Example
- 5 Conclusion





A fast, biochemistry-inspired, discrete event (meta-)simulator

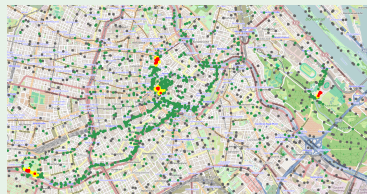
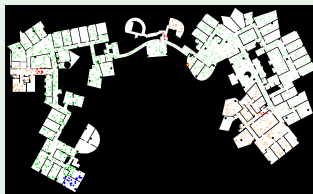
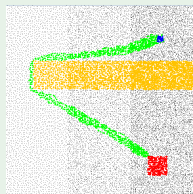
- Presented first here at MAS&S in 2011 :)
 - Now a much more sophisticated tool
- Written (mostly) in Java, simulation specifications in YAML
- Based on an extended version of Gibson-Bruck's kinetic Monte Carlo
- Defines a set of abstract entities (with names inspired by chemistry), whose actual implementation is left to the so-called incarnations
- Incarnations ready to use for:
 - The SAPERE EU Project [ZOA⁺15] (tuple manipulation in a chemical fashion across networks)
 - Protelis [PVB15] (Aggregate programming)
 - Multicellular biological systems (working, but under development)
- Scafi incarnation to be included soon



Typical simulated scenarios

- Many, possibly mobile nodes
- Non trivial environment setups

Screenshots



- Plant images can be used as indoor scenarios
- Support for simulating on OpenStreetMaps
- Support for navigating nodes along streets pedestrian, bike, mountain bike, motorcycle or car streets
- Support for GPS traces, with interpolation along streets



Alchemist distribution (shameless plug)

- Visit `alchemistsimulator.github.io`
 - Runnable JAR
 - Guides
 - Tutorial for Protelis at `https://github.com/AlchemistSimulator/Protelis-Incarnation-tutorial`

Gradle users

Import Alchemist from Maven Central in your build with:

```
compile 'it.unibo.alchemist:alchemist:+'
```

Maven users

Import Alchemist from Maven Central in your build with:

```
<dependency>  
  <groupId>it.unibo.alchemist</groupId>  
  <artifactId>alchemist</artifactId>  
  <version>LATEST</version>  
</dependency>
```

Alchemist and Scafi

The Scafi incarnation is ready and working

- Still, not distributed, we are waiting for Scafi to be released on Maven Central
- All the Alchemist major features are incarnation independent, and as such available immediately

Write once, simulate and deploy

- The simulator executes exactly the same Scafi code that will be deployed
 - Of course you can still break the toy by referencing internal Alchemist entities from Scafi via Java interoperability
 - But there's no cure for bad programming :)



Outline

- 1 Context and Motivation
- 2 Aggregate Programming
 - Basics
 - In Scala
- 3 Alchemist
- 4 Example
- 5 Conclusion

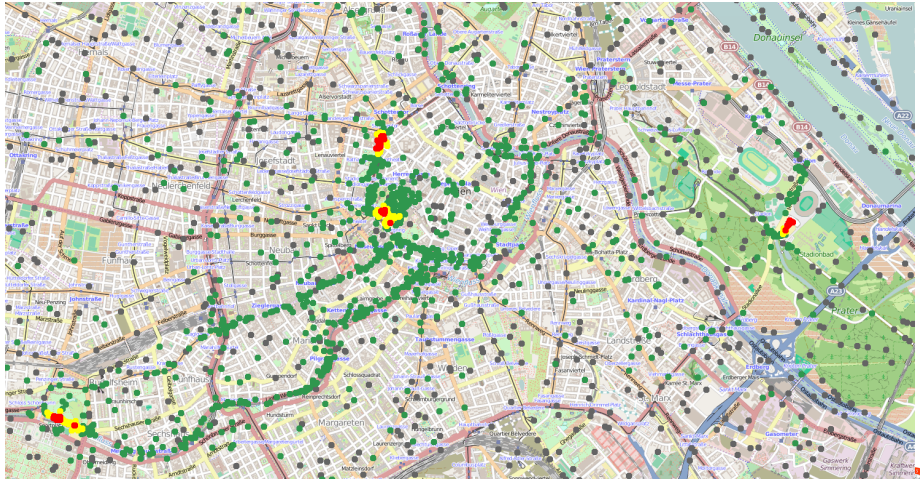


Crowd sensing and warning

- Real data from the Vienna City Marathon 2013
- We want to realise a system (potentially full P2P) which is able to detect areas where there is a dangerous crowd in the making
- We also want the system to warn those that have been close to such areas for a while



Case Study: crowd safety in a mass event



Scafi code I

Define three crowding levels

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

Locally estimate 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))
}
```



Scafi code II

Map each 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 }
}
```

S is a building block similar to G, C, and T that operates a division of the network.



Scafi code III

Measure if the dangerous situation held for long enough

```
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 }
}
```



Scafi code IV

Warn 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  
}
```



Alchemist simulation code I

Describe the environment and its geometry

```
environment:  
  type: OSMEnvironment  
  parameters: ["vcm.pbf", "vcmuser.agt", 12000]  
positions:  
  type: LatLongPosition
```

Describe the network connectivity model

```
network-model:  
  type: EuclideanDistance  
  parameters: [100]
```

Select the incarnation

```
incarnation: scafi
```



Alchemist simulation code II

Select which events will run, and their time distribution

pools:

- pool: `&program`
 - time-distribution: 1
 - program: `crowdWarning(0.005, 30, 60, 60)`
- pool: `&move`
 - time-distribution: 0.1
 - type: Event
 - actions:
 - type: GPSTraceWalker



Alchemist simulation code III

Displace the network nodes and program them

```
displacements:
- in:
  type: Rectangle
  parameters: [1479, 48.17, 16.3, 0.09, 0.15]
programs:
- *program
- *move
contents: []
```



Outline

- 1 Context and Motivation
- 2 Aggregate Programming
 - Basics
 - In Scala
- 3 Alchemist
- 4 Example
- 5 Conclusion



Conclusion I

Aggregate programming

- A promising approach for engineering large, situated systems
- A paradigm shift

Scafi

- A Scala implementation of the field calculus
- Strong, static type system
- Complex behaviours can be expressed compositionally
- Modern, concise, consistent syntax



Conclusion II

Contribution

We integrated Scafi (existing but very fresh Scala-based DSL for aggregate programming) and Alchemist (an existing simulator)

Scafi + Alchemist

- Behaviour verification in articulated environment
- Performance measurement
- Parameter tuning



References I



Danilo Pianini, Mirko Viroli, and Jacob Beal.

Protelis: practical aggregate programming.

In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853, 2015.



Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. De Angelis, Giovanna Di Marzo Serugendo, Simon A. Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, Stefano Mariani, Ambra Molesini, Sara Montagna, Jussi Nieminen, Danilo Pianini, Matteo Risoldi, Alberto Rosi, Graeme Stevenson, Mirko Viroli, and Juan Ye.

Developing pervasive multi-agent systems with nature-inspired coordination.

Pervasive and Mobile Computing, 17:236–252, 2015.



Implementation of G

```
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
```



Implementation of C

```
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 }
}
```



Implementation of T

```
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))
}
```



Implementation of S

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

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)
  }

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 } } }
    }
  }
}
```

