Programming (and Learning) Self-Adaptive & Self-Organising Behaviour with ScaFi

for Swarms, Edge-Cloud Ecosystems, and More

Roberto Casadei roby.casadei@unibo.it
Gianluca Aguzzi gianluca.aguzzi@unibo.it
Danilo Pianini danilo.pianini@unibo.it
Mirko Viroli mirko.viroli@unibo.it

Alma Mater Studiorum – Università di Bologna

Talk @ International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)

28/09/2023

Contents

Tutorial references

- ***Where do I start?***
  - `https://github.com/AggregateComputing/acsos-2023-scafi-tutorial`: repository with the examples/exercises proposed in the tutorial
    Follow the README.md to run examples and setup a development environment.
  - ScaFi-Web [1] (a online web simulator for ScaFi programs)
    `https://scafi.github.io/web`
    `https://tomcat-glad-muskox.ngrok-free.app/`

Frequently Asked Questions (FAQs) I

- ***What is it all about?*** ScaFi is an aggregate programming language: it supports the development of self-organising behaviours. It is integrated into Alchemist: this allows to build simulations of networked systems that execute ScaFi programs.
- ***Where do I start?*** Slide 3
- ***How can I quickly experiment with the programming model?***
  - ScaFi-Web [1] (a online web simulator for ScaFi programs)
    `https://scafi.github.io/web`
    `https://tomcat-glad-muskox.ngrok-free.app/`
- ***Where do I learn more about the tools (ScaFi, Alchemist)?***
  - `https://scafi.github.io/`
  - `http://alchemistsimulator.github.io/`
- ***Where do I learn more about the theory/research on aggregate computing (AC)?*** Check out relevant presentations and papers.
  - `https://www.slideshare.net/RobertoCasadei/aggregate-computing-research-an-overview` – a quick overview of AC research
  - M. Viroli, J. Beal, F. Damiani, *et al.*, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019. doi: 10.1016/j.jlamp.2019.100486 – a survey about AC, its history, and main developments
- ***What is the research context?*** Check out Slide 7–Slide 15
- ***What kind of systems can be programmed/developed with AC?*** Check out Slide 20 and Slide 7–Slide 15

Contents

Contents

Context

Modern IT systems are more and more **complex**

- increasing availability of wearable/mobile/embedded/flying devices
- increasing availability of heterogeneous wireless networks
- increasing availability of computational resources (edge/fog/cloud computing)
- increasing production of data **everywhere** and **anytime**

The challenge

Consider the worst-case possible scenario

- **zillion** of devices *unpredictably* located and moving in a space
- heterogeneous displacement, **pervasive** sensing/actuation
- computational services are **contextual** (proximity-based) and *dynamic*

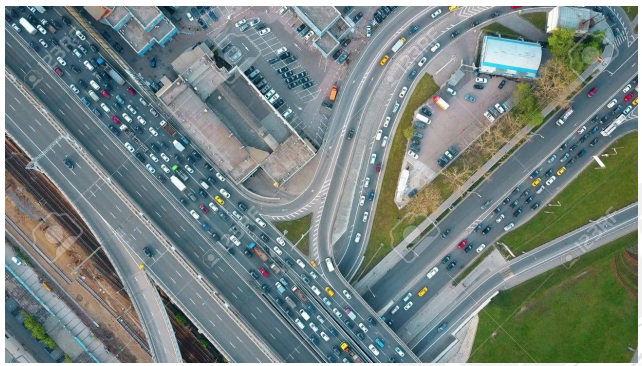
How can we **program** such systems?

What are the right **abstractions**?

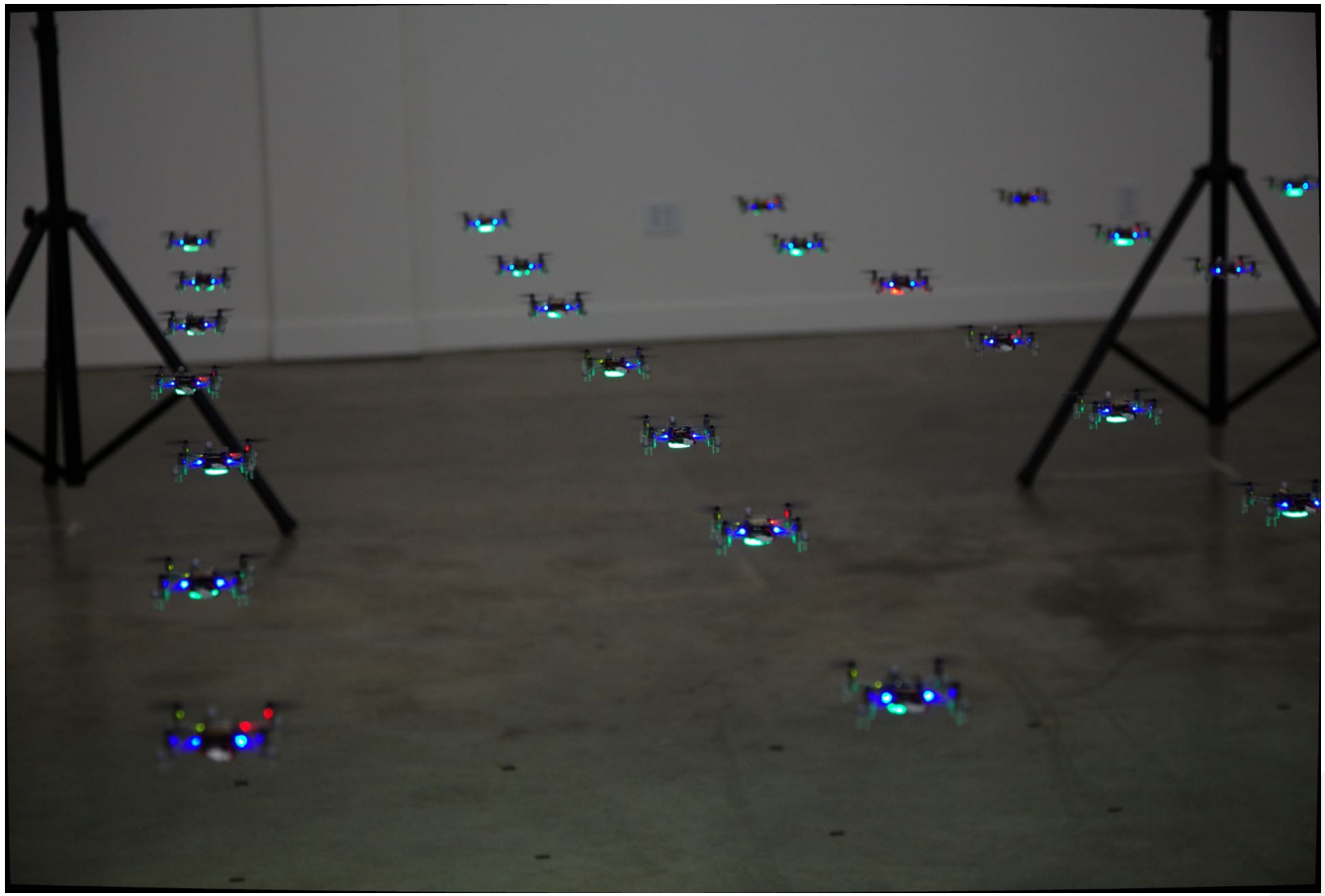
Collective Adaptive Systems (CASs)

Systems composed of *possibly* **large** set of component executing a **collective** task strongly relying on component **interactions** and showing **inherent** *adaptivity*.





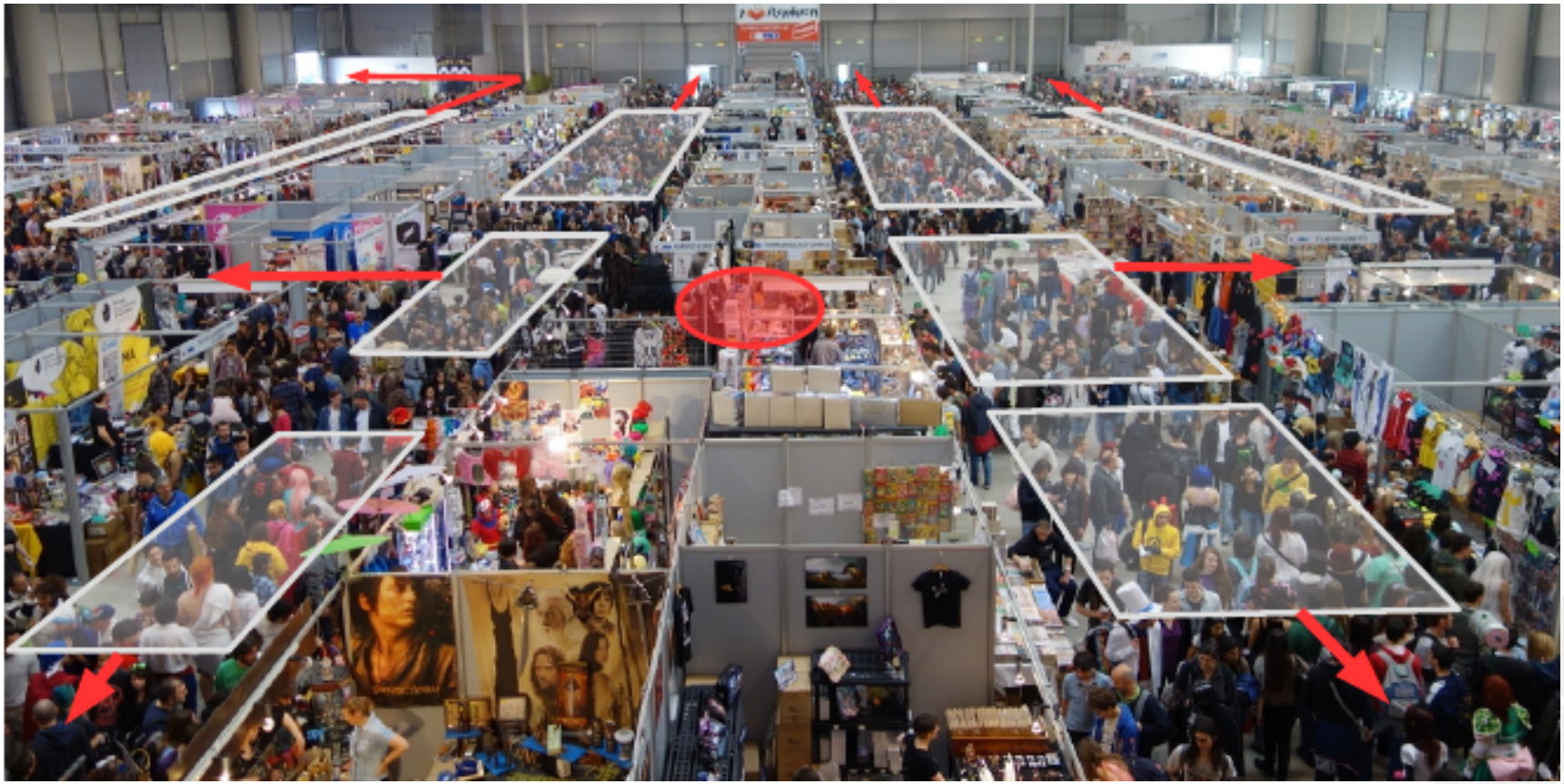
Examples: controlling a fleet of drones



Issues

- Design techniques to **reactively** propagate information across the fleet
- Create algorithms for **higher-level** programming of the fleet

Examples: pedestrian steering in (smart)cities



Issue

- Design **self-organisation** algorithms for people navigation
- More generally, design map-based large-scale applications

Examples: fine control of crowds

Issues

- Evaluating the influence of environment in crowd formation
- Evaluating the influence of ambient sensors, cameras and wearable devices

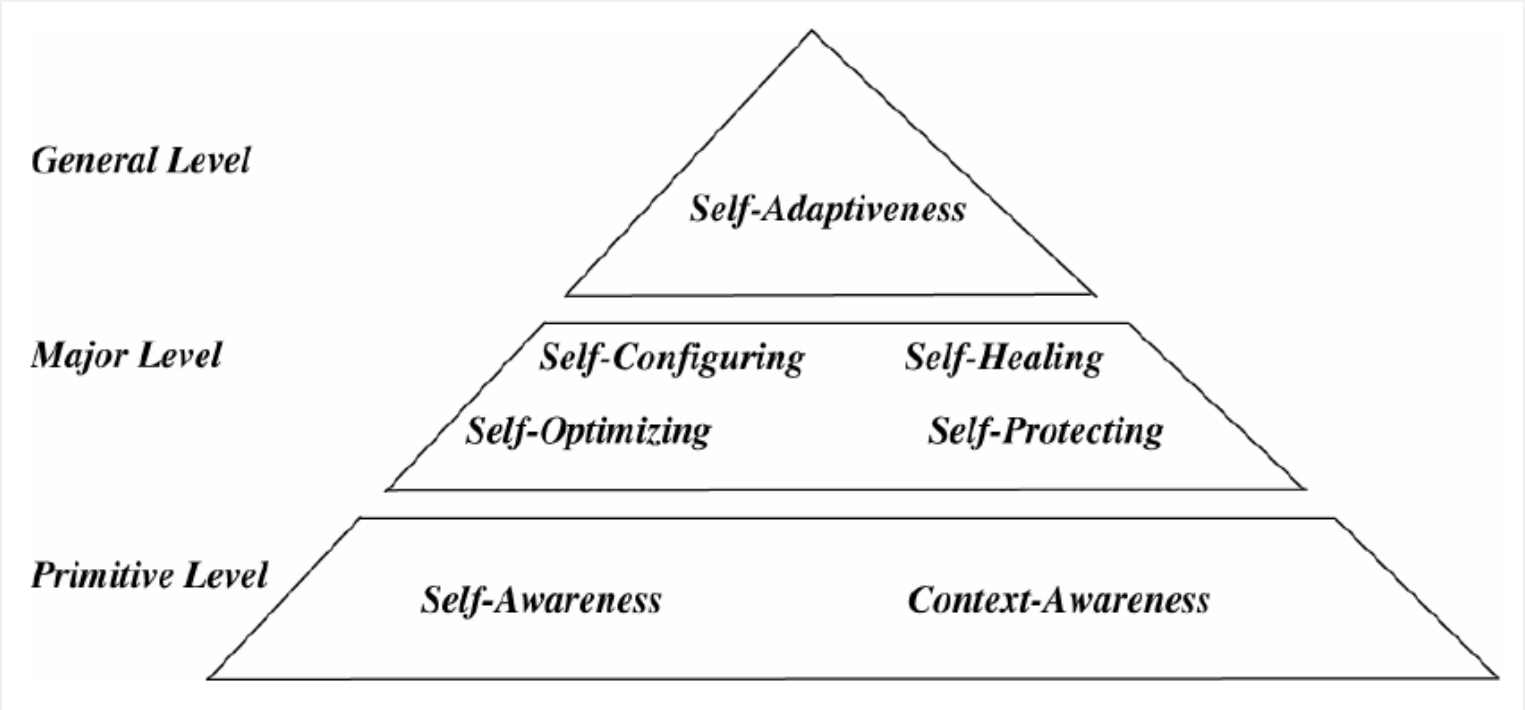
Collective Adaptive Systems

Adaptation

A system is *adaptive* if it can **change** its behaviour depending on **circumstances**, in order to better reach its *goal*

➔ Adaptivess in **intrisic** to open complex system (like CASs)

Kind of adaptiveness, self-* properties



Kind of adaptiveness

Self-adaptiveness: general level

- **self-adaptiveness** ➔ the ultimate property we perceive from outside
- the terms "self-*" recalls a property achieved in **autonomy**
- subcases: self-managing, -governing, -maintenance, -control

Major level

- internal properties related to overall system management
- subcases: self-configuring, self-healing, self-optimising, self-protecting
- defined in the context of **autonomic** computing

Primitive level

- internal properties related to primitive aspects
- subcases: self-awareness, context-awareness
- means being able to properly perceive the own state, context, ...

The case of self-organisation

Self-organisation

The ability of creating **spatial/temporal** patterns out of the *local interaction* of individuals

Self-adaptive vs. self-organising

- self-adaptive: a top-down way of achieving adaptiveness

➔ we have the adaptation goal, and accordingly guide components

- self-organising: a bottom-up way of achieving adaptiveness

➔ the adaptive behaviour emerges from local interactions

  - The typical approach in large-scale CASs
  - Born in the context of **swarm intelligence**

Good Abstraction for CASs?

Ideas

- specify **overall** behaviour, not *individual* device program
- abstract from the actual shape of **interactions**
- **automatically** adapt to environment details
- focus on how the **global output** pattern can be obtained from global inputs
- focus on both **spatial** and **temporal** computing patterns

➔ the ideas around macro-programming paradigms [3]!

  - **Aggregate computing**
  - Buzz
  - ...

- Ruccurent abstractions:
  - Ensembles & collective tasks
  - Self-organising information flows
  - Self-healing collective structures (e.g., *gradients*)

---

[3] R. Casadei, "Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling," *ACM Comput. Surv.*, vol. 55, no. 13s, 2023, issn: 0360-0300. doi: 10.1145/3579353

Contents

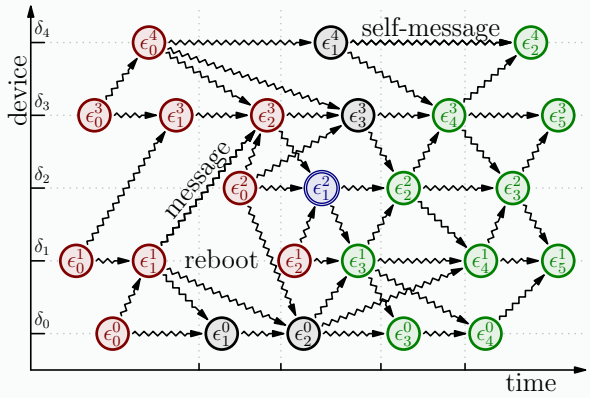
Aggregate Computing (AC) in 1 Slide

Self-org-like computational model

interaction: *repeated* msg exchange with **neighbours**

behaviour: *repeated* execution of async rounds of **sense – compute – (inter)act**

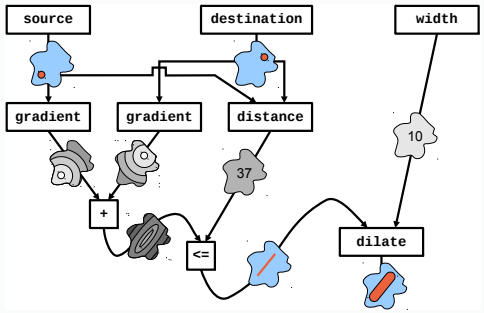
formal model of executions: **event structures**



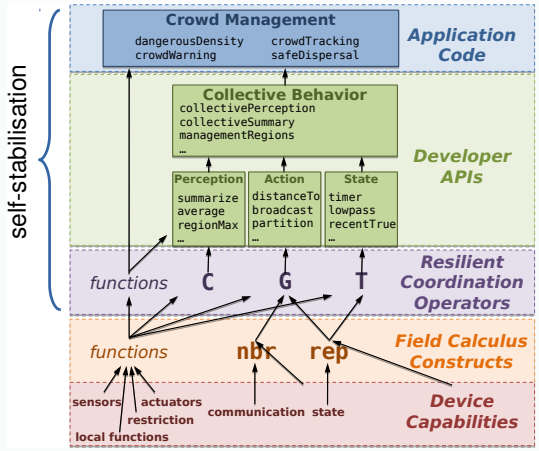
abstraction: **computational fields** ($dev/evt \mapsto \mathbb{V}$)

formal core language: **field calculus [2]**

paradigm: **functional, macro-programming**

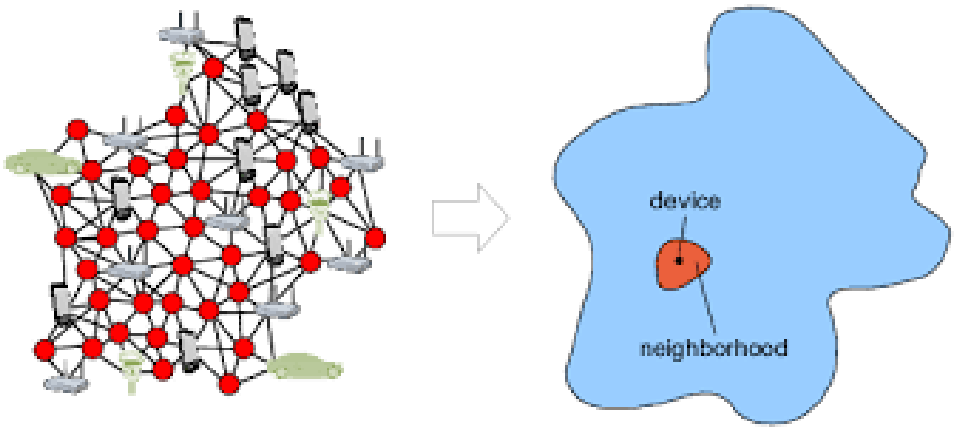



```
def channel(source: Boolean, target: Boolean, width: Double) =
  dilate(gradient(source) + gradient(target) <=
    distance(source, target), width)
```



M. Viroli, J. Beal, F. Damiani, *et al.*, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019. doi: 10.1016/j.jlamp.2019.100486

J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015. doi: 10.1109/MC.2015.261



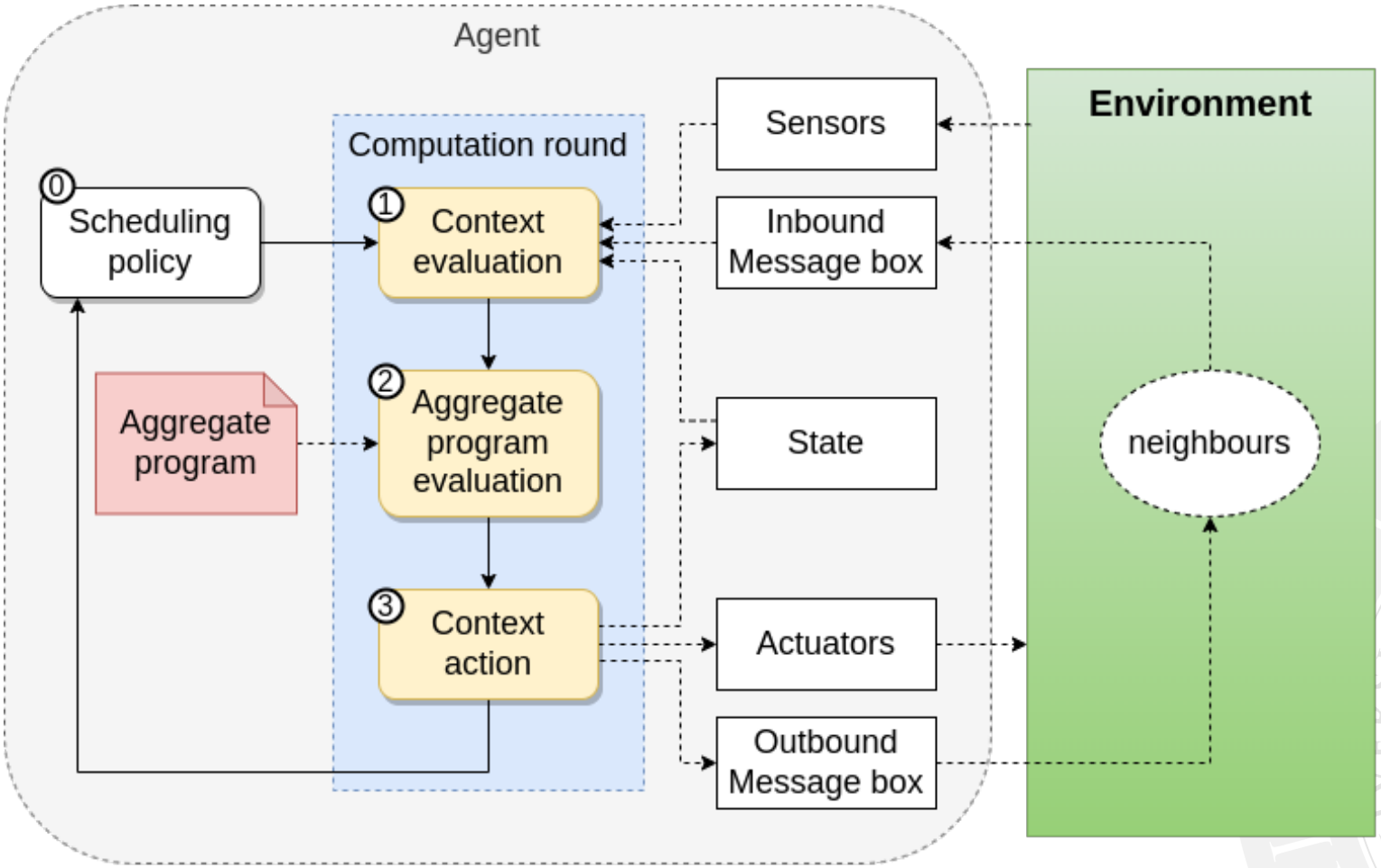
Aggregate Computing

Program the **aggregate**, not the individual device

- *Computing Machine* ➔ an ensemble of devices as a single body, fading the actual space
- *Elaboration process* ➔ atomic manipulation of a *collective* data structure (**computational field**)
- *Networked computation* ➔ a proximity-based self-organising system hidden "under-the-hood"

Computational model – self-organisation-like execution

- *Continuous* communication with **neighbours** only (➔decentralisation)
- *Continous* execution of async **rounds** of sense - compute - communicate



Applicability: in a nutshell

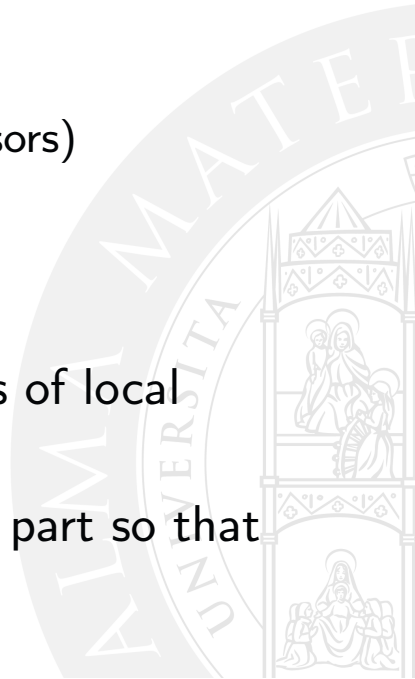
Essentially, this approach can be used to program any system that can be recast to the following

- **Structure:** network of locally interacting agents
  - neighbouring relationship: e.g. based on network connectivity, physical distance, social relationship, or whatever
  - dynamicity/openness: devices may move/fail, enter/exit the system at runtime, and neighbourhoods may change
- **Interaction:**
  - **with neighbours:** asynchronous message passing
  - **with environment:** via sensors and actuators
- **Behaviour:** sense–compute–interact
  - **sense:** acquire context (messages from neighbours, values from sensors)
  - **compute:** mapping context to (inter-)action
  - **interact:** sending messages to neighbours and running actuation

So, best for applications that are:

- progressive/long-running: require multiple coordinated steps/rounds of local processing and action

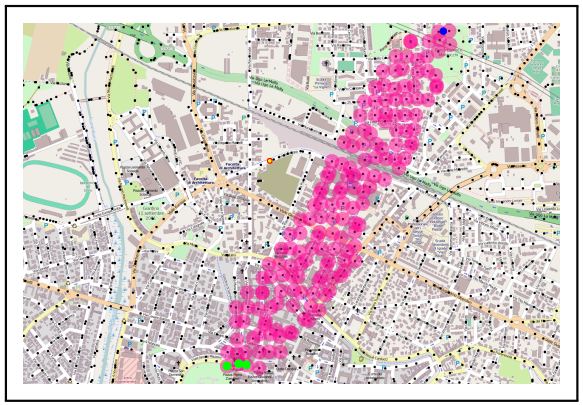
Assuming this system model, AC is about how to specify the "compute" part so that global emergent outcomes can be eventually attained



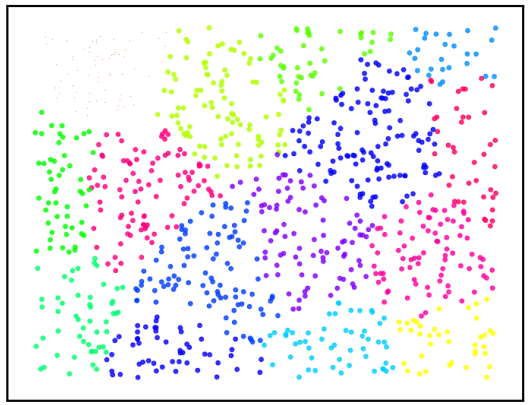
Computational Fields: a static view

Traditionally a map: *Space* $\mapsto$ *Values*

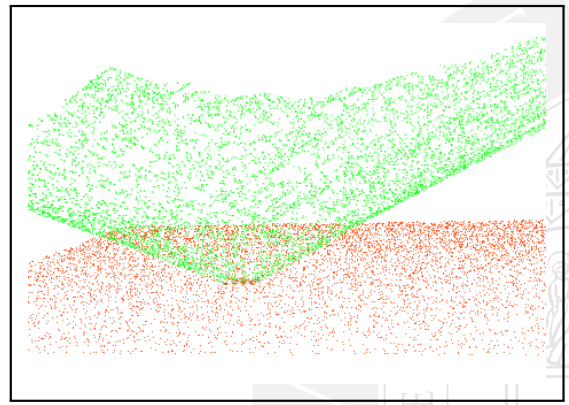
- possibly: evolving over time, dynamically injected, stabilising
- smoothly adapting to very heterogeneous domains
- more easily *"understood"* on continuous and flat spatial domains
- ranging to booleans, reals, vectors, functions



boolean channel in 2D



numeric partition in 2D



real-valued gradient in 3D

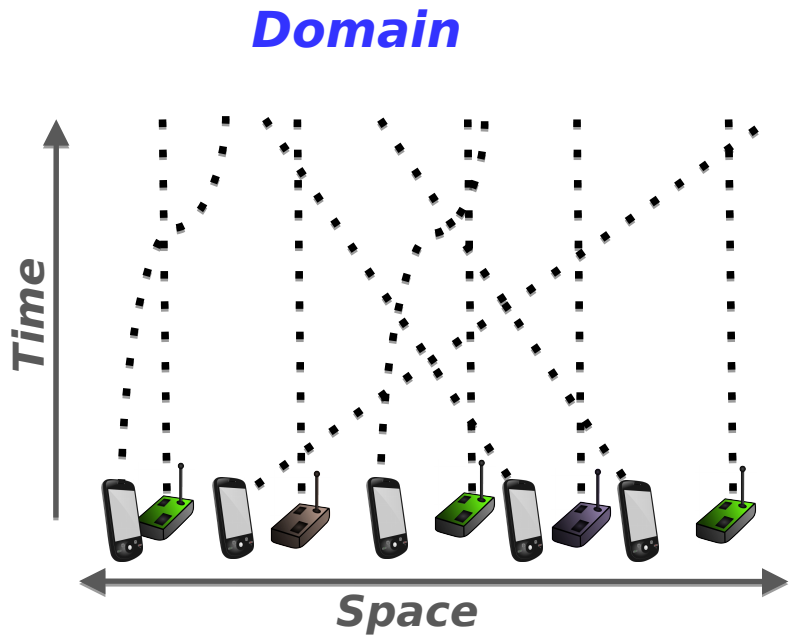
Computational Fields revisited: a dynamic view

A field as a *space-time* structure: $\phi : D \mapsto V$

- *event* E: a triple $\langle \delta, t, p \rangle$ – device δ, "firing" at time t in position p
- *events domain* D: a coherent set of events (devices cannot move too fast)
- *field values* V: any data value

➜ computation abstracts from/adapts to the underlying event

➜ scheduling of events is essentially exogenous



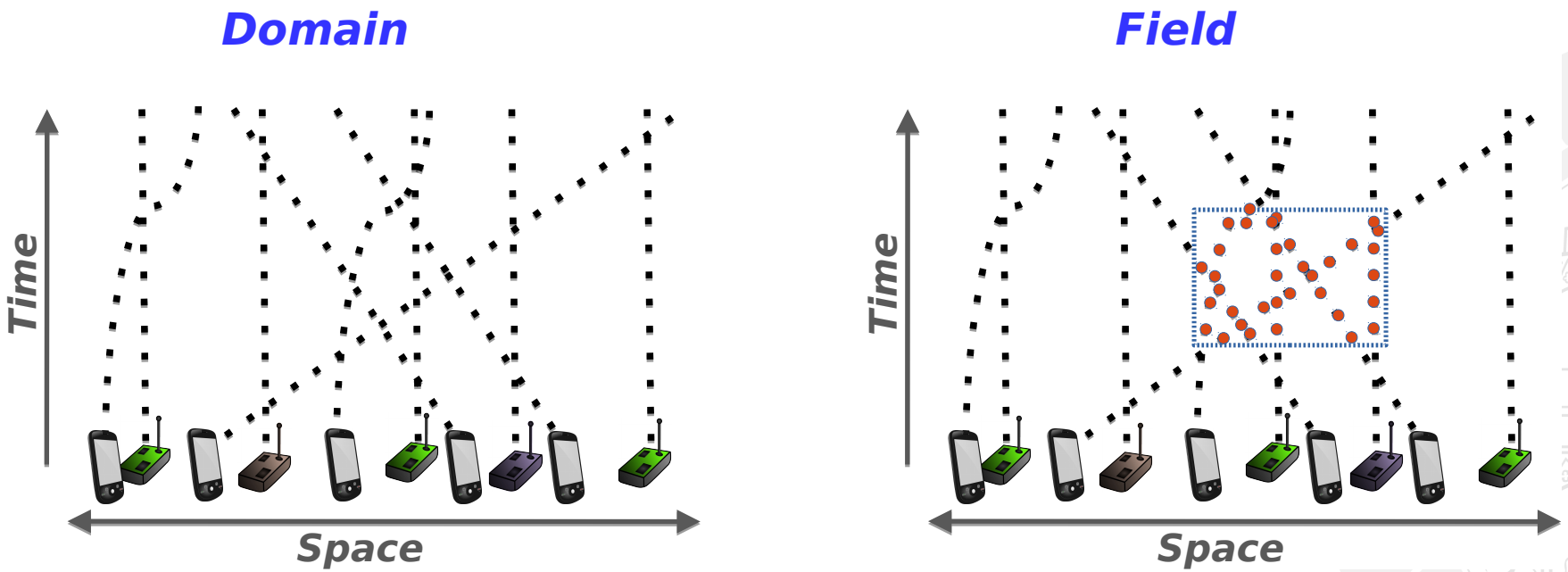
Computational Fields revisited: a dynamic view

A field as a *space-time* structure: $\phi : D \mapsto V$

- *event* E: a triple $\langle \delta, t, p \rangle$ – device δ, "firing" at time t in position p
- *events domain* D: a coherent set of events (devices cannot move too fast)
- *field values* V: any data value

➜ computation abstracts from/adapts to the underlying event

➜ scheduling of events is essentially exogenous



Computational Fields revisited: a dynamic view

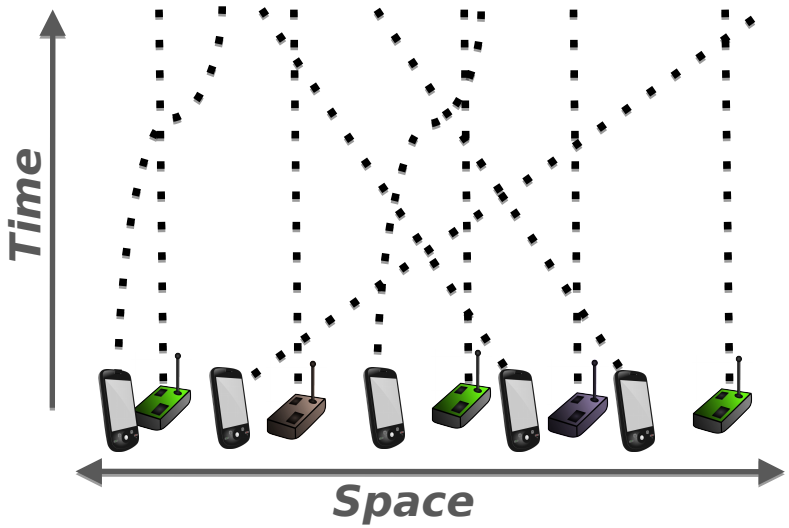
A field as a *space-time* structure: $\phi : D \mapsto V$

- *event* E: a triple $\langle \delta, t, p \rangle$ – device δ, "firing" at time t in position p
- *events domain* D: a coherent set of events (devices cannot move too fast)
- *field values* V: any data value

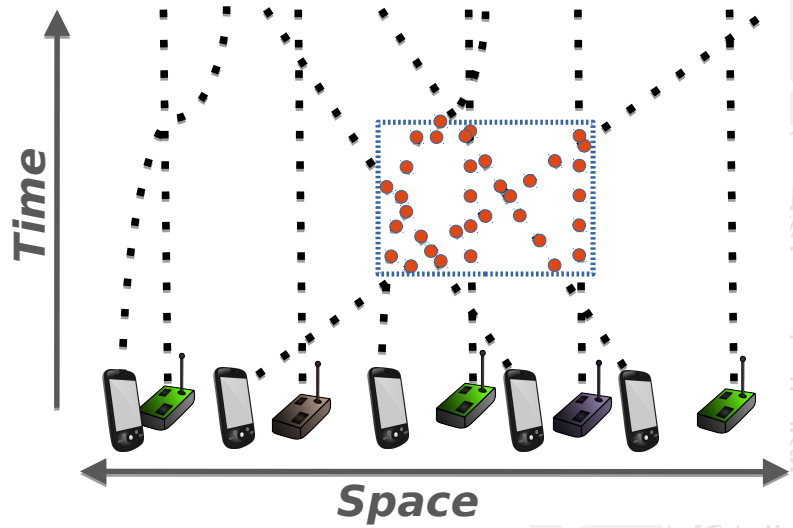
➔ computation abstracts from/adapts to the underlying event

➔ scheduling of events is essentially exogenous

Domain



Field



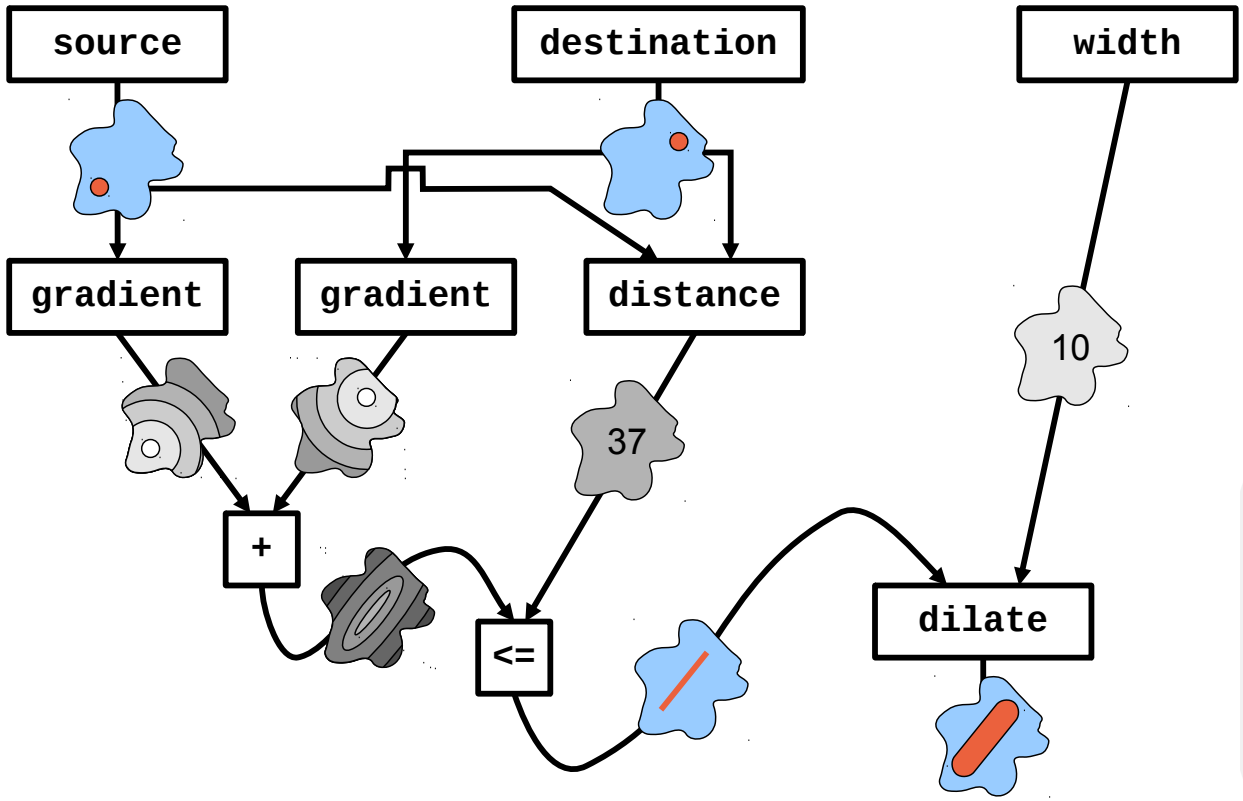
will later show only snapshots of fields in 2D space..

Aggregate programming as a functional approach

Functionally composing fields

- **Inputs**: sensor fields, Output: actuator field
- Computation is a pure function over fields (time embeds state!)

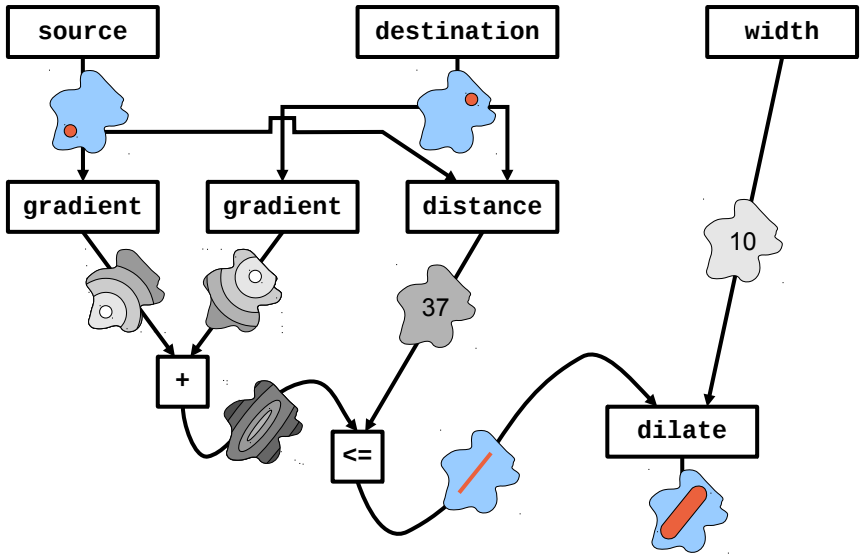
➜ for this to be practical/expressive we need a good programming language



Preview

How do we want that computation to be expressed?

- `source`, `dest` and `width` as inputs
- `gradient`, `distance` and `dilate` as reusable functions
- ➜ note we are reusing and composing global-level, aggregate specs




```
def channel(source: Boolean, dest: Boolean, width: Double): Boolean = {
  dilate(gradient(source) + gradient(dest) <= distance(source,dest), width)
}
```

Field calculus model

Key idea

- a sort of λ-calculus with "everything is a field" philosophy!

Syntax – Reduced

$$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid \texttt{e}(\texttt{e}_1, \ldots, \texttt{e}_n) \mid \texttt{rep}(\texttt{e}_0)\{\texttt{e}\} \mid \texttt{nbr}\{\texttt{e}\} \quad \text{(expr)}$$

$$\texttt{v} ::= < \text{standard-values} > \mid \lambda \quad \text{(value)}$$

$$\lambda ::= \texttt{f} \mid \texttt{o} \mid (\overline{\texttt{x}})\texttt{=>e} \quad \text{(functional value)}$$

$$\texttt{F} ::= \texttt{def f}(\overline{\texttt{x}})\ \{\texttt{e}\} \quad \text{(function definition)}$$

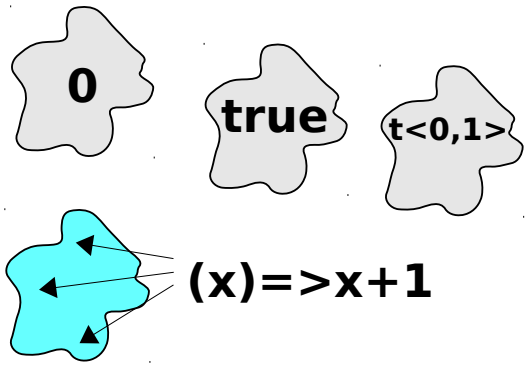
Few explanations

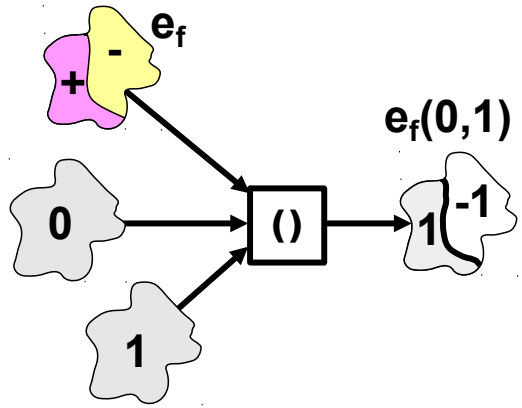
- `v` includes numbers, booleans, strings,..
  ..tuples/vectors/maps/any-ADT (of expressions)
- `f` is a user-defined function (the key aggregate computing abstraction)
- `o` is a built-in local operator (pure math, local sensors,..)

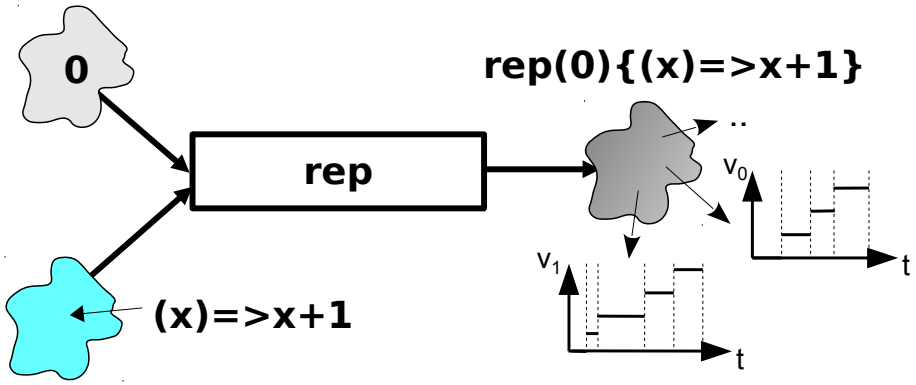
Intuition of global-level (denotational) semantics

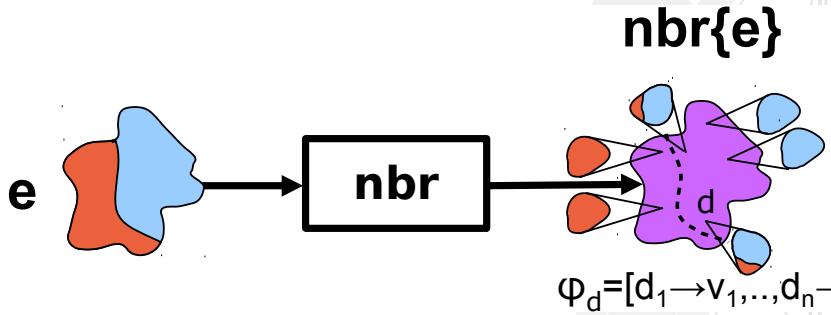
The four main constructs at work
⇒ values, application, evolution, and interaction – in aggregate guise

- $e ::= \ldots \mid v \mid e(e_1, \ldots, e_n) \mid \texttt{rep}(e_0)\{e\} \mid \texttt{nbr}\{e\}$









Intuition of global-level semantics – More

Value $\mathtt{v}$

- A field constant in space and time, mapping any event to $\mathtt{v}$

Function application $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$

- $\mathtt{e}$ evaluates to a field of functions, assume it ranges to $\lambda_1, \ldots, \lambda_n$
- this naturally induces a partition of the domain $D_1, \ldots, D_n$
- now, join the fields: $\forall i$, $\lambda_i(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ restricted in D_i

Repetition $\mathtt{rep}(\mathtt{e}_0)\{\mathtt{e}_\lambda\}$

- the value of $\mathtt{e}_0$ where the restricted domain "begins"
- elsewhere, unary function $\mathtt{e}_\lambda$ is applied to previous value at each device

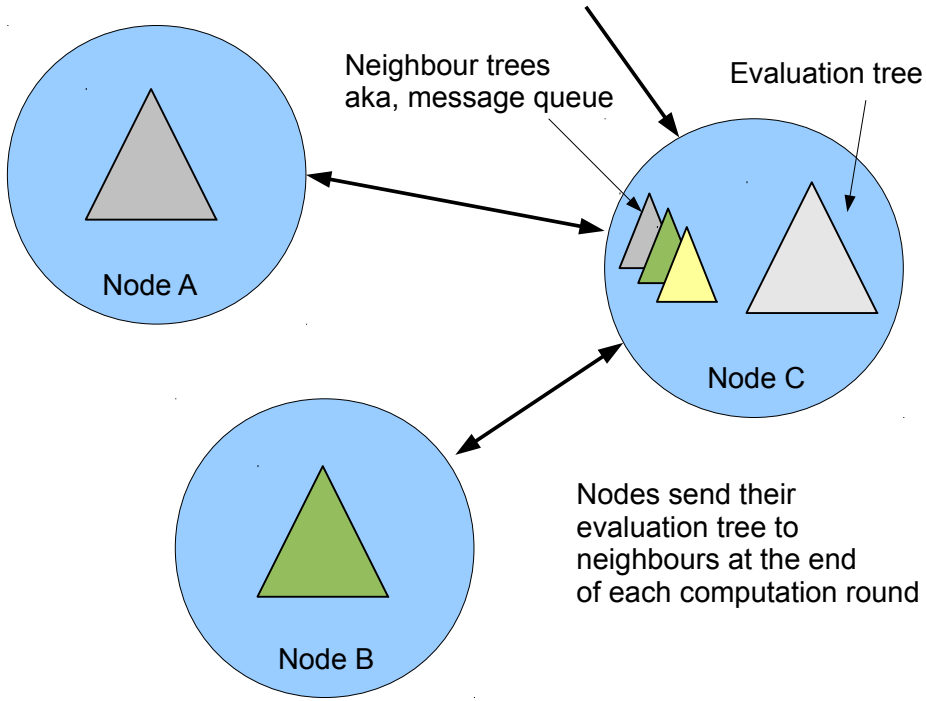
Neighbouring field construction $\mathtt{nbr}\{\mathtt{e}\}$

- at each event gathers most recent value of $\mathtt{e}$ in neighbours (in restriction)
- ..what is neighbour is orthogonal (i.e., physical proximity)

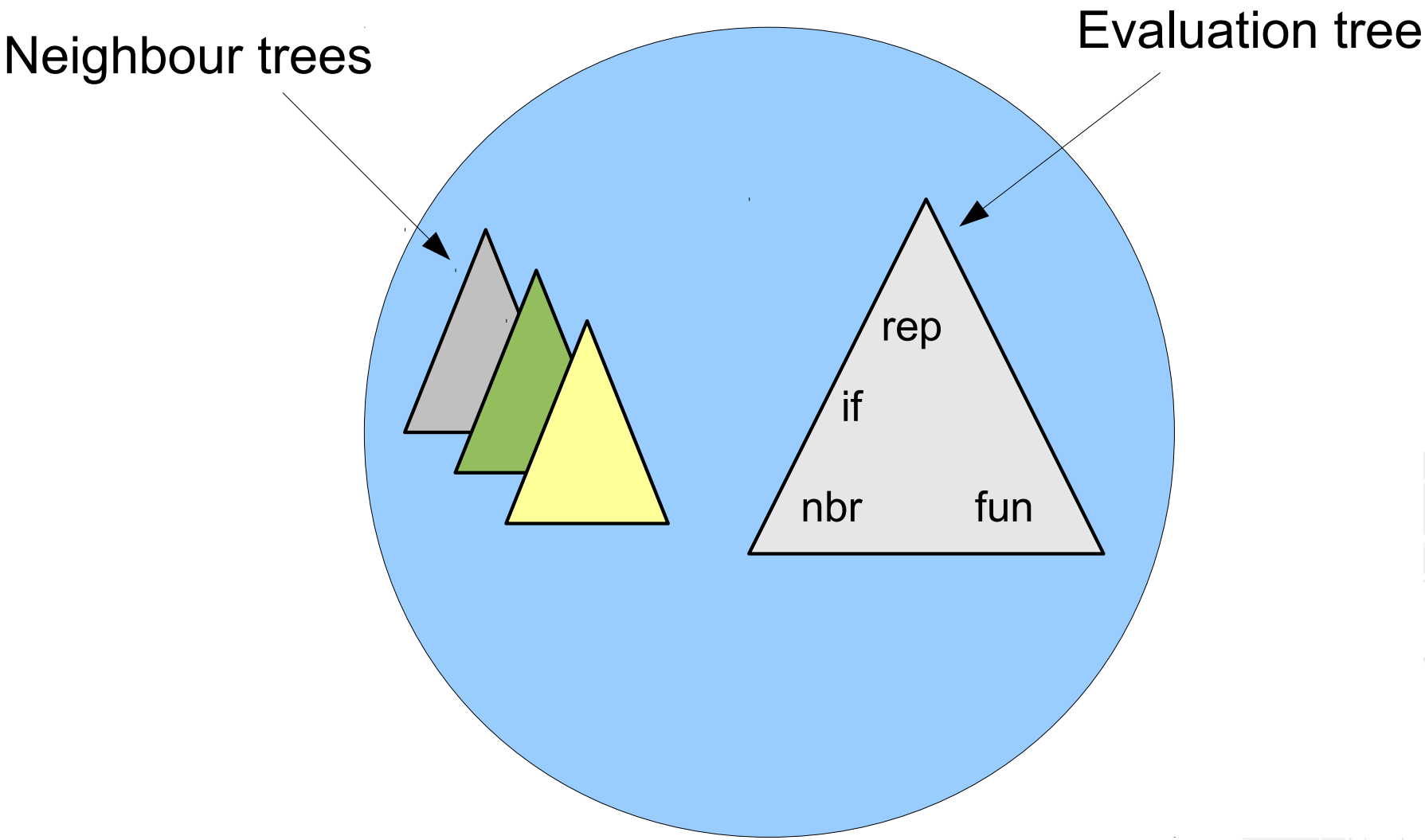
Key aspects of the semantics: network model

Platform abstract model

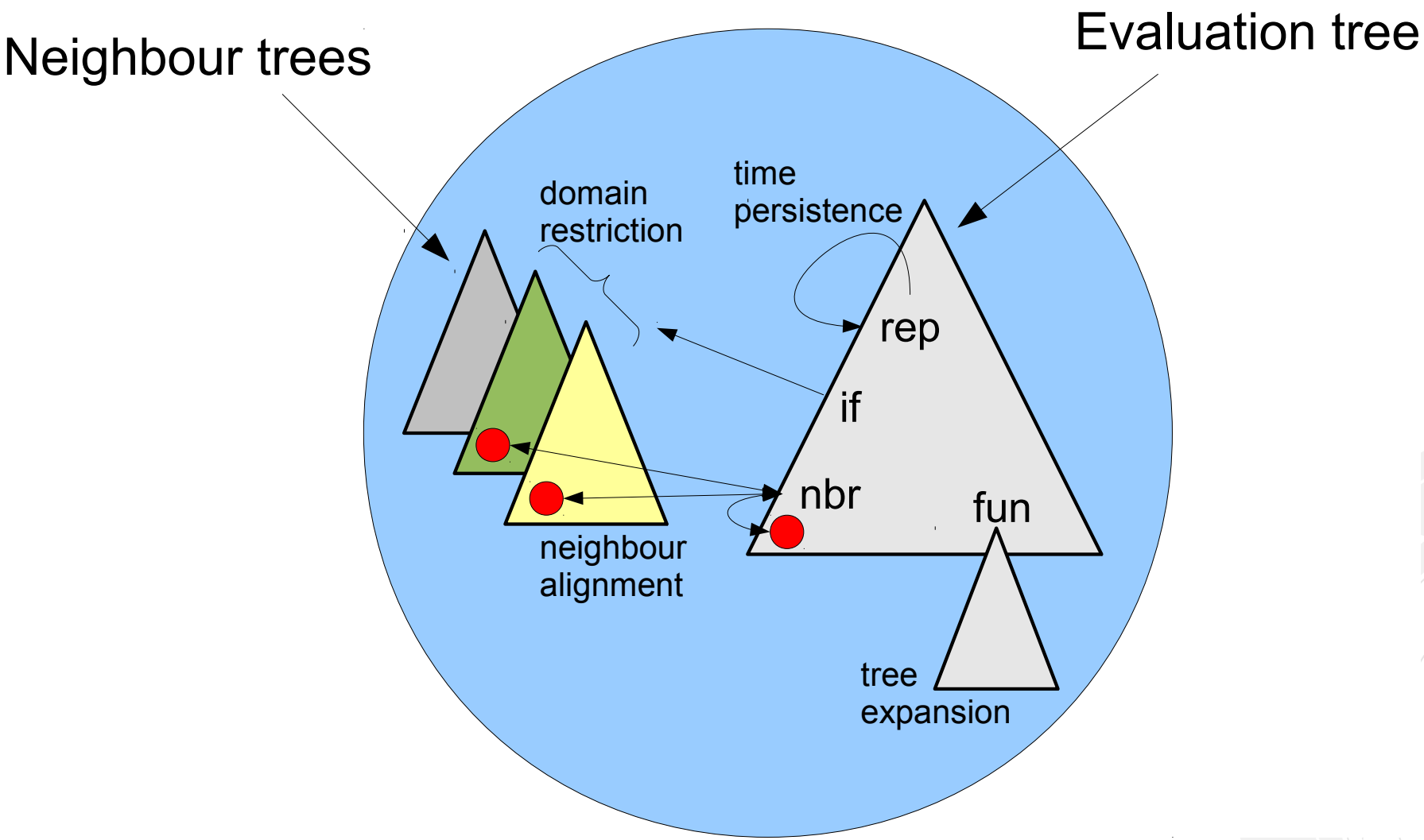
- A node state θ (value-tree) updated at asynchronous rounds
- At the end of the round, θ is made accessible to the neighbourhood
- A node state is updated "against" recently received neighbours' trees



Tree evaluation: pictorial semantics



Tree evaluation: pictorial semantics



Core mechanisms in the operational semantics

Orthogonally..

- evaluation proceeds recursively on expression and neighbour trees
- neighbour trees may be discarded on-the-fly if not "aligned" (restriction)

Core mechanisms in the operational semantics

Orthogonally..

- evaluation proceeds recursively on expression and neighbour trees
- neighbour trees may be discarded on-the-fly if not "aligned" (restriction)

Function application $\mathtt{e}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$

- evaluates the body against a filtered set of neighbours ...
- ..i.e., only those which evaluated e to the same result

Repetition $\mathtt{rep}(\mathtt{e}_0)\{\mathtt{e}_\lambda\}$

- if a previous value-tree of mine is available, evaluates $\mathtt{e}_\lambda$ on its root
- otherwise, evaluates $\mathtt{e}_0$

Neighbouring field construction $\mathtt{nbr}\{\mathtt{e}\}$

- gather values from neighbour trees currently aligned
- add my current evaluation of e

Operational semantics as a blueprint for platform support

Requirements

- a notion of the neighbourhood must be defined — wireless connectivity, physical proximity ...
- nodes execute in asynchronous rounds, and emit a "round result"
- a node needs to have recent round results of neighbours
- by construction we tolerate losses of messages
- by construction we tolerate various round frequencies

Operational semantics as a blueprint for platform support

Requirements

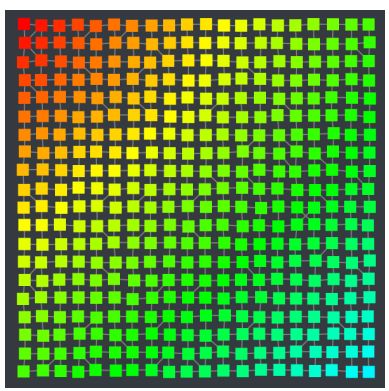
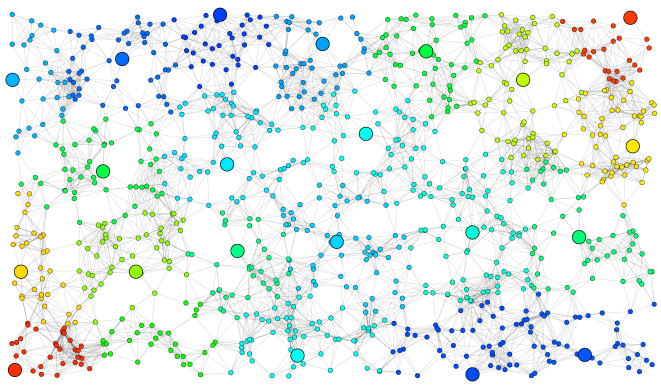
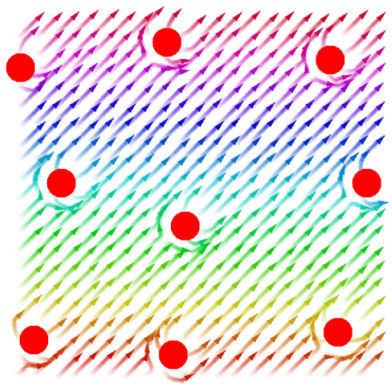
- a notion of the neighbourhood must be defined — wireless connectivity, physical proximity ...
- nodes execute in asynchronous rounds, and emit a "round result"
- a node needs to have recent round results of neighbours
- by construction we tolerate losses of messages
- by construction we tolerate various round frequencies

Platform details are very orthogonal to our programming model!

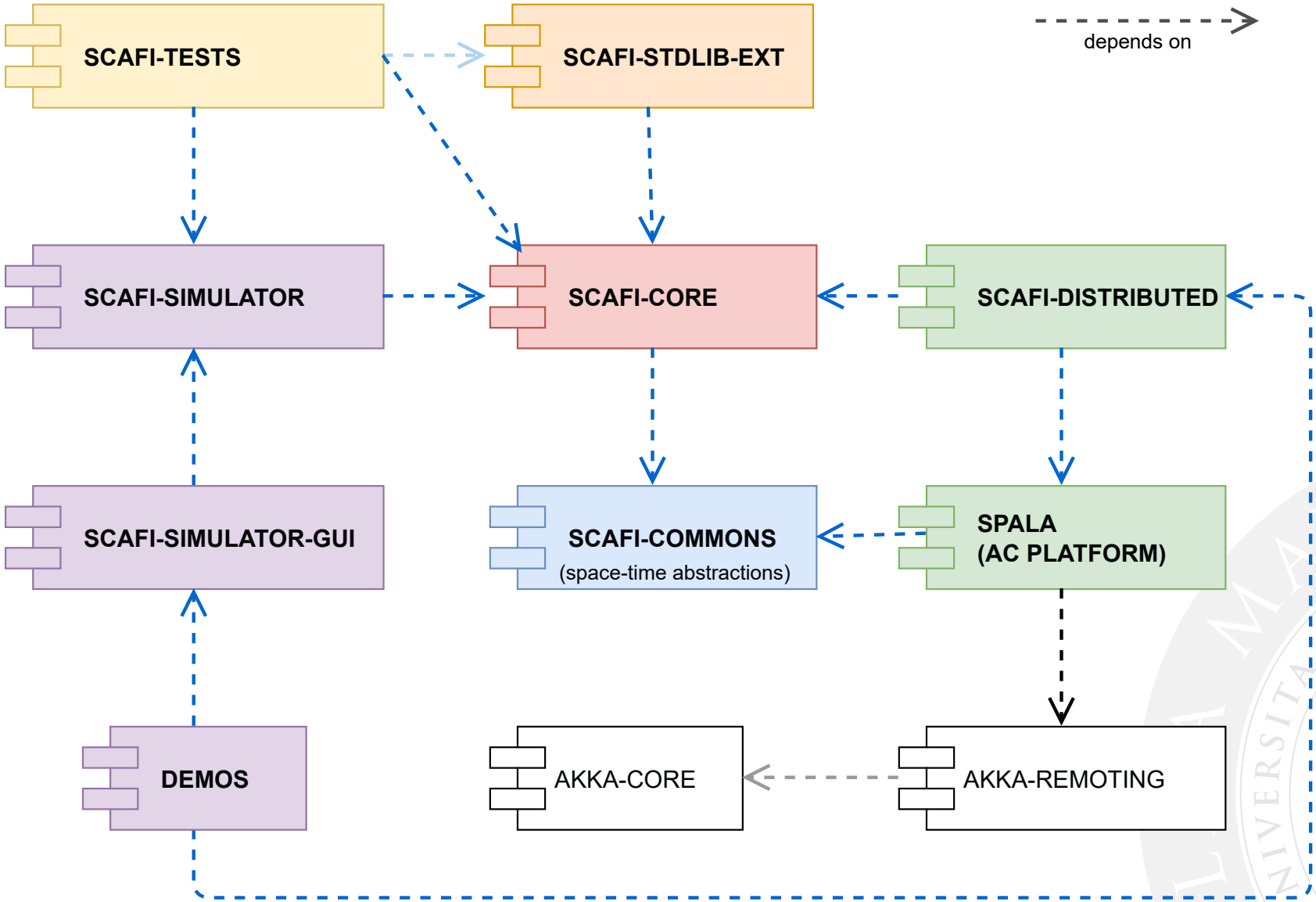
- the above requirements can be met by various platforms
- *programming remains mostly unaltered!*

ScaFi (Scala Fields)

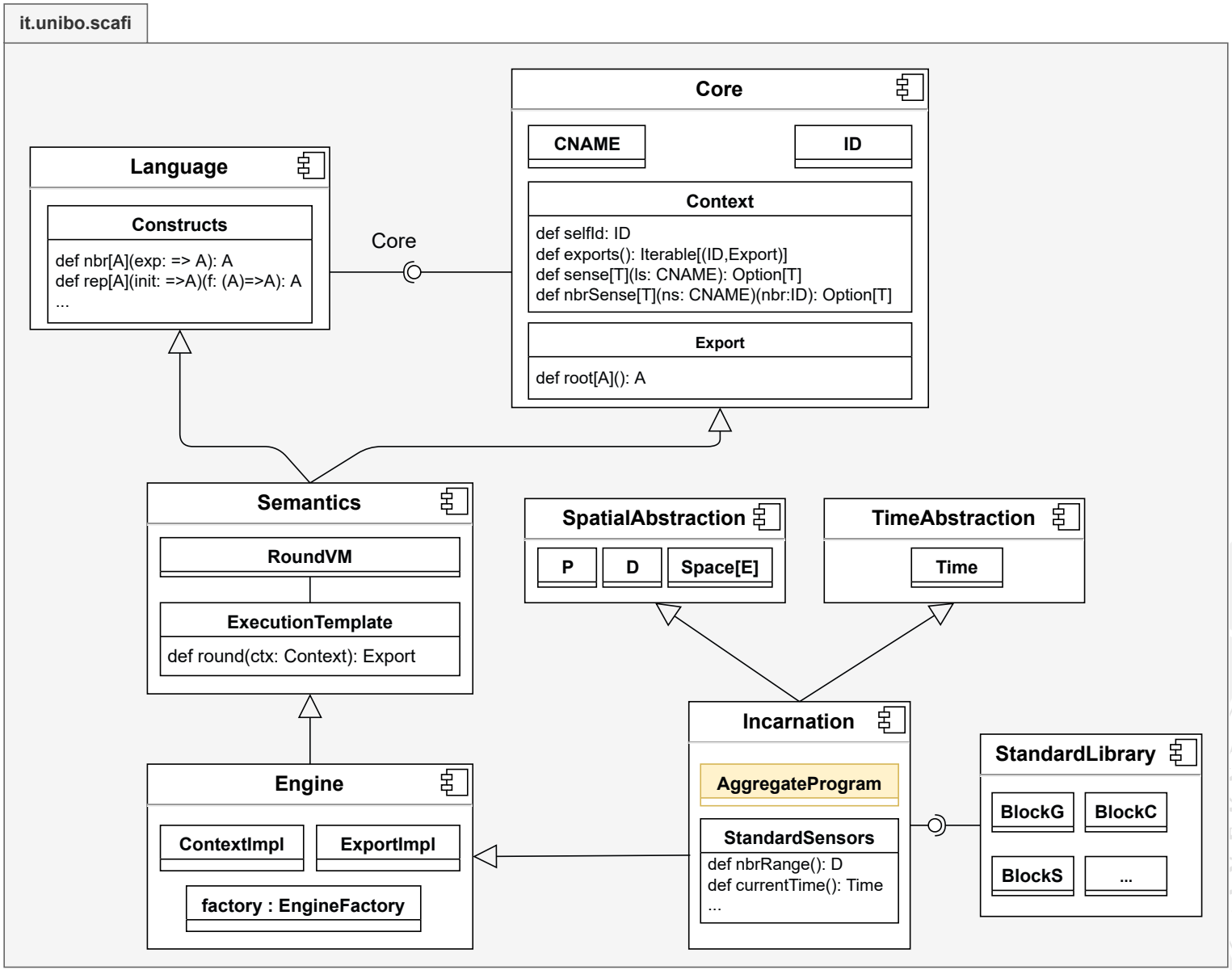
A Scala toolkit providing an *internal domain-specific* language, *libraries*, a *simulation* environment, and *runtime* support for **practical** aggregate computing systems development

ScaFi: Organization



ScaFi: Design



ScaFi: syntax as a core language/API

```
trait Constructs {
  // the unique identifier of the local device
  def mid(): ID

  // applies fun to the previous result, or to init at the first call
  def rep[A](init: A)(fun: (A) => A): A

  // evaluation of expr at the currently-considered neighbour
  def nbr[A](expr: => A): A

  // accumulates available evaluations of expr, with acc/init monoid
  def foldhood[A](init: => A)(acc: (A,A)=>A)(expr: => A): A

  // splits computation: th where cond is true, el everywhere/time else
  def branch[A](cond: => Boolean)(th: => A)(el: => A): A

  // perception of local sensor
  def sense[A](name: CNAME): A

  // perception of neighbourhood sensor
  def nbrvar[A](name: CNAME): A
  ...
}
```

ScaFi: setup an aggregate application

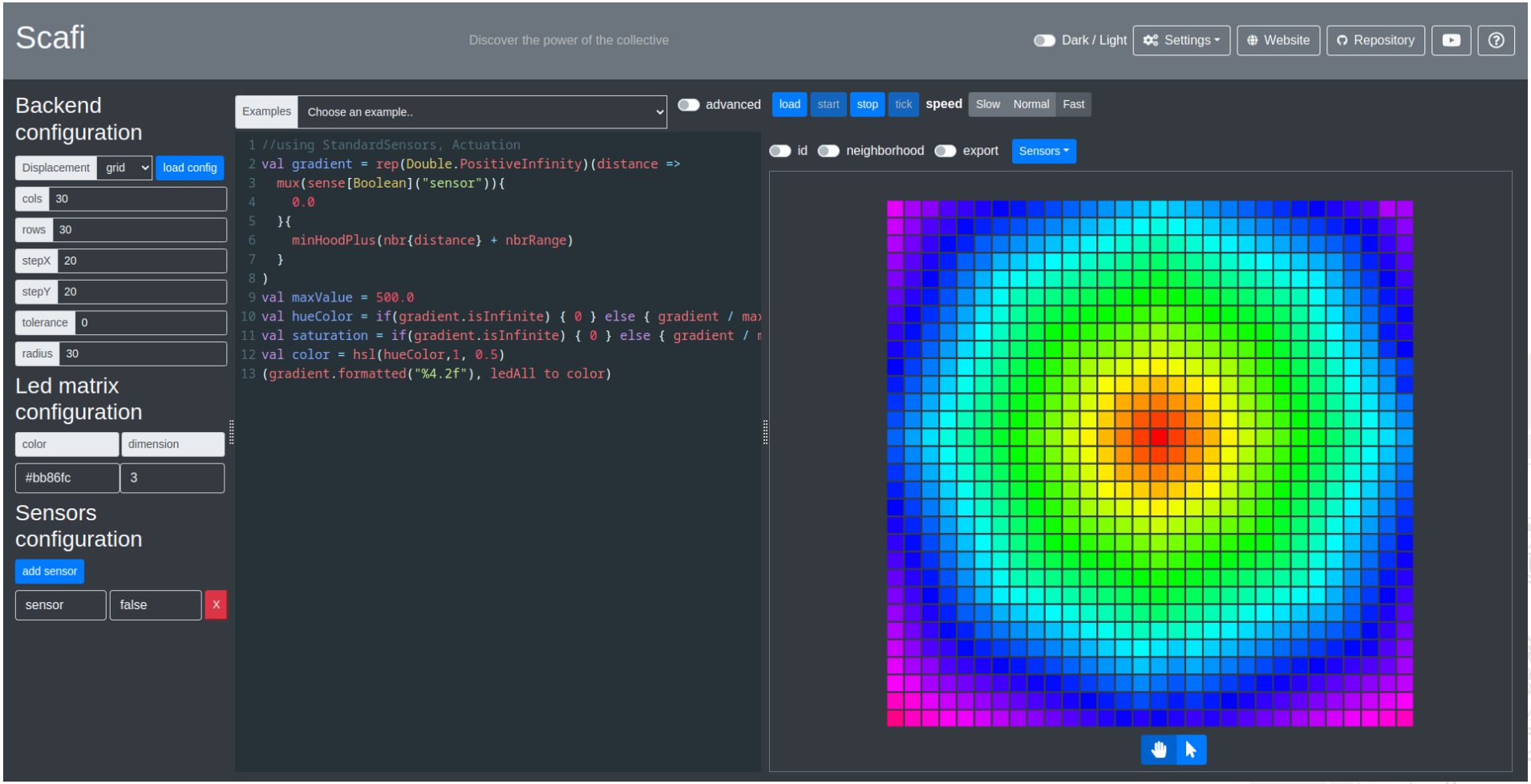
```
package experiments
// STEP 1 Choose an incarnation (simulated or real)
import it.unibo.scafi.incarnations.BasicSimulationIncarnation._

// STEP 2 Define the aggregate program including the right libraries
class MyProgram extends AggregateProgram with Libs {
  // STEP 2.1 Define main logic of the program
  override def main(): Any = ...
}

// STEP 3 Platform/Node setup (both in simulation/real)

// STEP 3.1 in case of ScaFi simulation:
object SimulationRunner extends Launcher {
  Settings.Sim_ProgramClass = "experiments.MyAggregateProgram"
  Settings.ShowConfigPanel = true
  launch()
}
```

So let us start playing with ScaFi!! ☺

Contents

An aggregate computing playground – ScaFi Web!

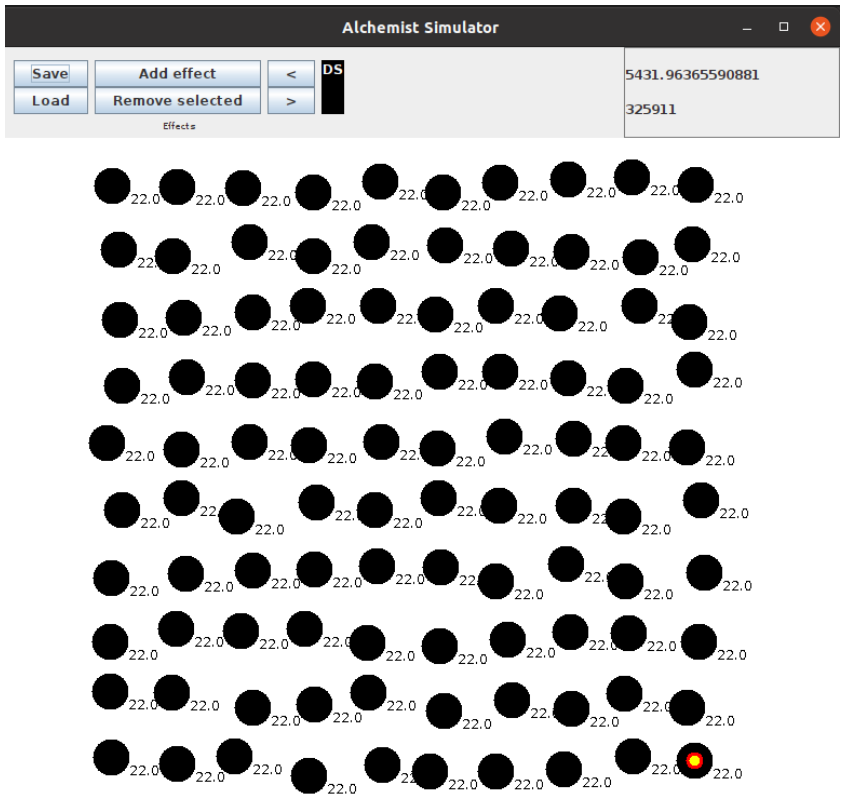
https://scafi.github.io/web/



Guided Examples!

Example 1: static constant field

```
// 1. Select the incarnation
import it.unibo.alchemist.model.scafi.ScafiIncarnationForAlchemist._

// 2. define your aggregate program by extending 'AggregateProgram'
class Example1 extends AggregateProgram {
  // 3. define the main method of the aggregate computing script
    override def main(): Int = 22
}
```



Example 3: sensor query

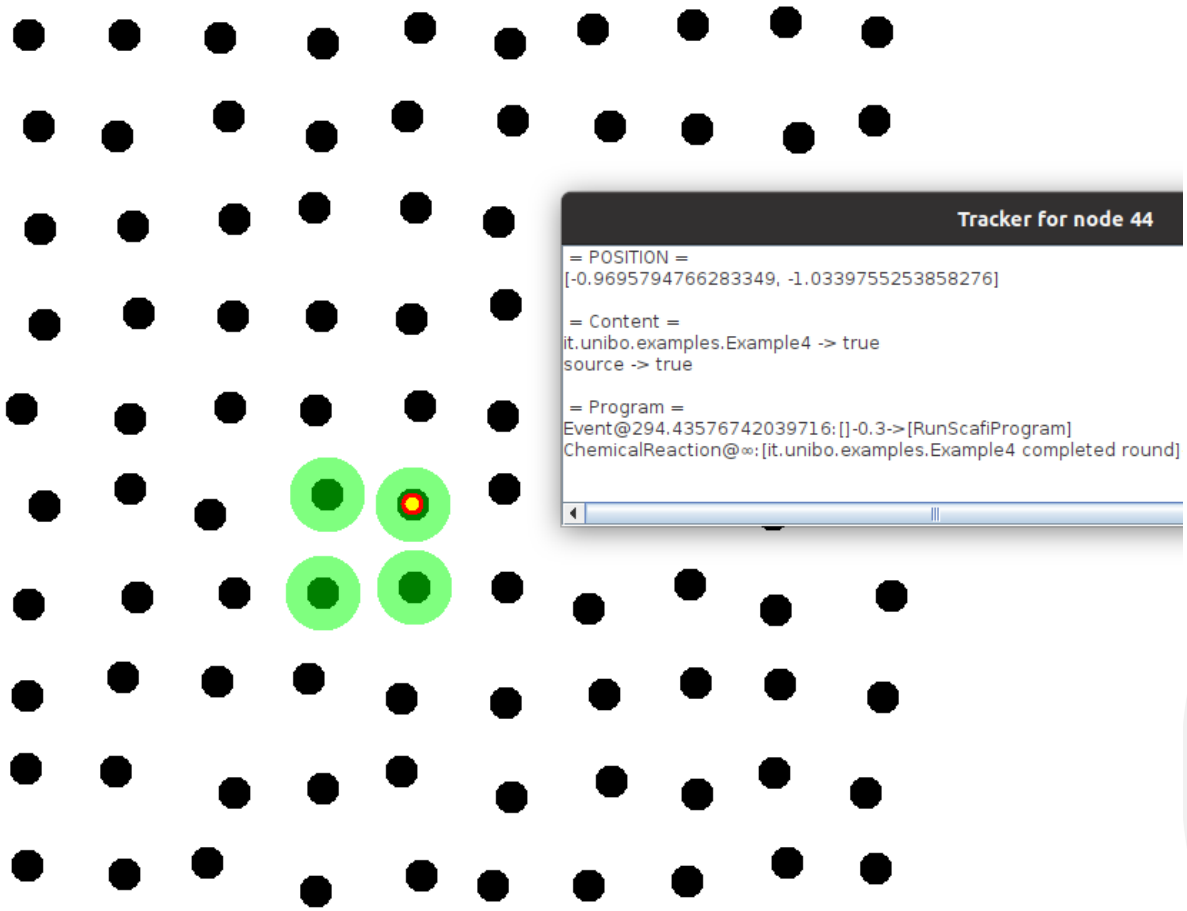
`mid()` provides the device ID

```
// Code in the AggregateProgram's main method:
mid()
```

Example 4: sensor query

`sense("name")` reads the current value of a sensor

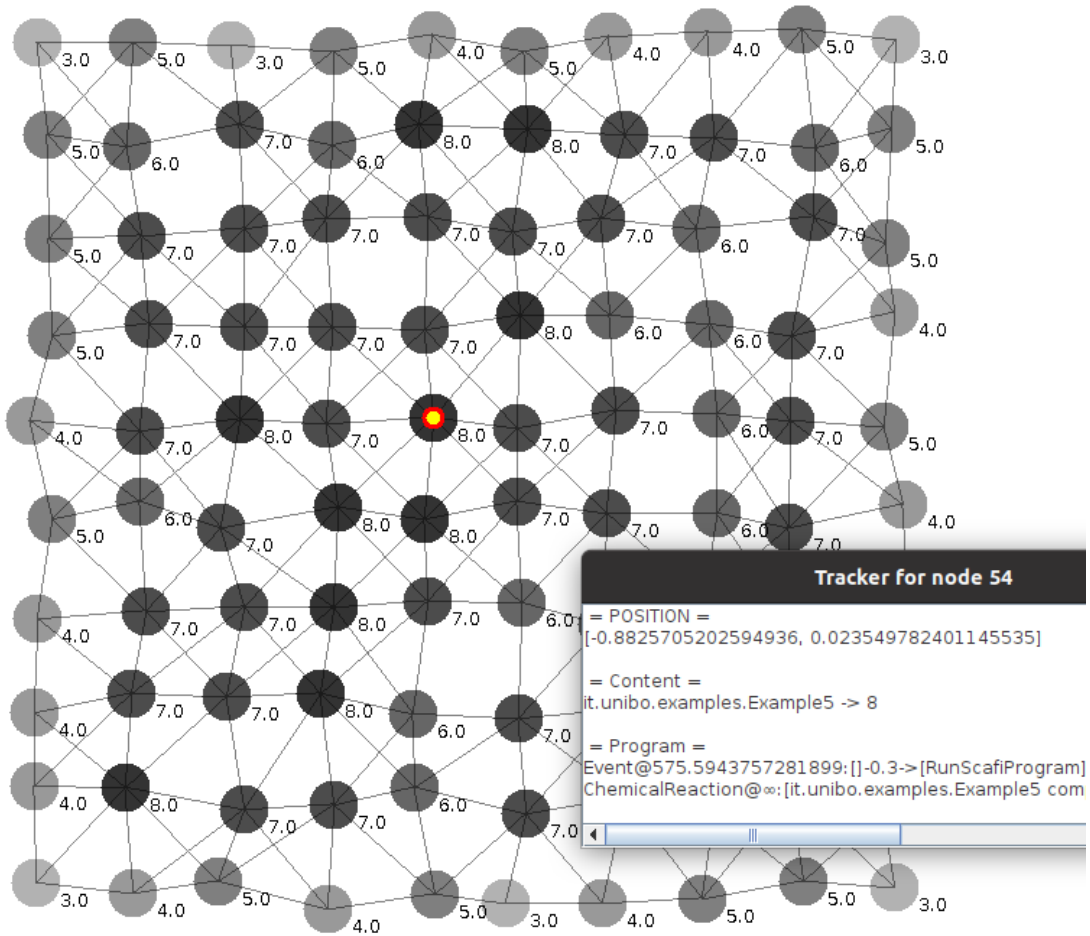
```
sense[Boolean]("source")
```



Example 5: neighbour interaction

`foldhood(i)(f)(e)` aggregates with `f` the neighbours' values for `e`

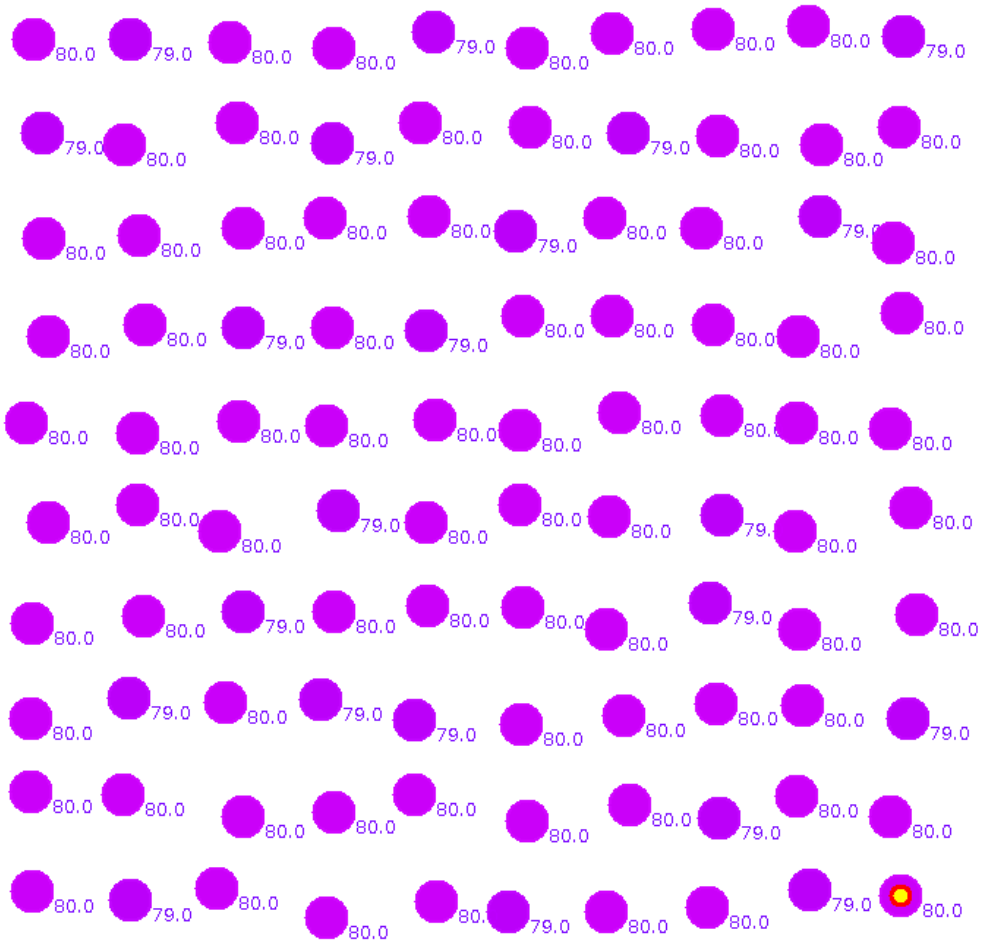
```
// *Plus version does not consider the device itself in the neighbourhood
foldhoodPlus(0)((a,b) => a + b)(nbr(1))
```



Example 6: stateful computations

`rep(i)(f)` updates a value (initially `i`) through function `f`

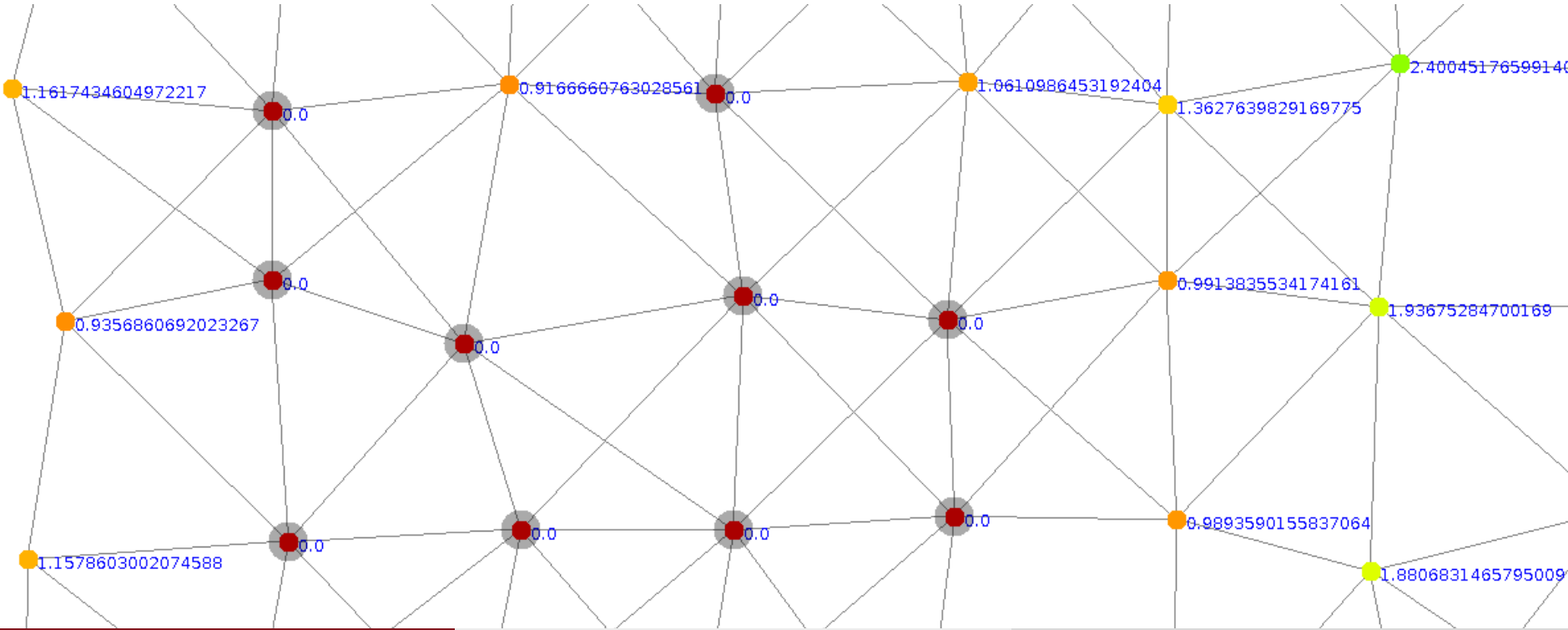
```
rep(0){ _ + 1 }
```



Example 8: self-healing gradient

Field of minimum distances from source nodes

```
rep(Double.PositiveInfinity)(distance =>
    mux(sense[Boolean]("source")) {
      0.0
    } {
      minHoodPlus(nbr(distance) + nbrRange)
    }
  )
```



Programming (and Learning) Self-Adaptive & Self-Organising Behaviour with ScaFi

for Swarms, Edge-Cloud Ecosystems, and More

Roberto Casadei roby.casadei@unibo.it
Gianluca Aguzzi gianluca.aguzzi@unibo.it
Danilo Pianini danilo.pianini@unibo.it
Mirko Viroli mirko.viroli@unibo.it

Alma Mater Studiorum – Università di Bologna

Talk @ International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)

28/09/2023

References I

[1] G. Aguzzi, R. Casadei, N. Maltoni, D. Pianini, and M. Viroli, "Scafi-web: A web-based application for field-based coordination programming," in *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings*, F. Damiani and O. Dardha, Eds., ser. Lecture Notes in Computer Science, vol. 12717, Springer, 2021, pp. 285–299. doi: 10.1007/978-3-030-78142-2_18. [Online]. Available: https://doi.org/10.1007/978-3-030-78142-2_18.

[2] M. Viroli, J. Beal, F. Damiani, G. Audrito, R. Casadei, and D. Pianini, "From distributed coordination to field calculus and aggregate computing," *J. Log. Algebraic Methods Program.*, vol. 109, 2019. doi: 10.1016/j.jlamp.2019.100486.

[3] R. Casadei, "Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling," *ACM Comput. Surv.*, vol. 55, no. 13s, 2023, issn: 0360-0300. doi: 10.1145/3579353.

[4] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, 2015. doi: 10.1109/MC.2015.261.