2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS) | 979-8-3503-3744-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ACSOS58161.2023.00026

Self-Organisation Programming: A Functional Reactive Macro Approach

Roberto Casadei, Francesco Dente, Gianluca Aguzzi, Danilo Pianini, Mirko Viroli
Department of Computer Science and Engineering
Alma Mater Studiorum–Università di Bologna, Cesena, Italy
{roby.casadei, gianluca.aguzzi, danilo.pianini, mirko.viroli}@unibo.it
{francesco.dente}@studio.unibo.it

***Abstract*—Engineering self-organising systems – e.g., robot swarms, collectives of wearables, or distributed infrastructures – has been investigated and addressed through various kinds of approaches: devising algorithms by taking inspiration from nature, relying on design patterns, using learning to synthesise behaviour from expectations of emergent behaviour, and exposing key mechanisms and abstractions at the level of a programming language. Focussing on the latter approach, most of the state-of-the-art languages for self-organisation leverage a round-based execution model, where devices repeatedly evaluate their context and control program fully: this model is simple to reason about but limited in terms of flexibility and fine-grained management of sub-activities. By inspiration from the so-called functional reactive paradigm, in this paper we propose a *reactive self-organisation programming approach* that enables to fully decouple the program logic from the scheduling of its sub-activities. Specifically, we implement the idea through a functional reactive implementation of aggregate programming in Scala, based on the functional reactive library Sodium. The result is a functional reactive self-organisation programming model, called FRASP, that maintains the same expressiveness and benefits of aggregate programming, while enabling significant improvements in terms of scheduling controllability, flexibility in the sensing/actuation model, and execution efficiency.**

***Index Terms*—self-organising systems, self-organisation programming, multi-agent systems, emergent behaviour, emergence steering, functional reactive programming, aggregate computing.**

I. Introduction

Building *artificial self-organising systems* exhibiting *collective intelligence* is a relevant research challenge spanning science and engineering [1]–[4]. A central problem lies in *driving the (emergence of) self-organising behaviour of a collection of agents or devices*—a goal also referred to through terms like “guided self-organisation” [5], “controlled self-organisation” [6], and “emergence steering” [7], [8]. This problem can be reduced to the definition of the control program that each agent has to execute [9]. The problem can be addressed through *automatic* approaches (e.g., multi-agent reinforcement learning [10]) or *manual* approaches [9] based on the definition of control rules or designs in terms of patterns involving, e.g., information flows and control loops [11]. Hybrid approaches are also possible where (parts of) programs are generated or improved through learning [12].

In this work, we focus on the *programming language-based* approach to self-organisation, where the developer writes the self-organising control program using a suitable *macroprogramming language* [13], [14] (i.e., one aiming at expressing the macro-level behaviour of a system), be it general-purpose or domain-specific (e.g., explicitly tailored to robotic swarms [15] or wireless sensor networks [16]). Specifically, our high-level goal is to devise a programming language for self-organising systems that is *expressive*, *practical*, and *declarative*—in the sense that it should allow the programmer to abstract from many operational details, to be dealt with automatically by the underlying middleware/platform [17], [18]. Specifically, we focus on the problem of concrete scheduling of the sub-activities of which a self-organising system can be composed. State-of-the-art languages typically leverage a *round-based* execution model, where devices repeatedly evaluate their context and control program entirely (typically in a loop or periodic, time-driven fashion). This approach is simple to reason about but limited in terms of flexibility in scheduling and management of sub-activities (and response to contextual changes). Motivated by this, and inspired by the functional reactive paradigm, in this work *we propose a reactive self-organisation programming language that enables the decoupling of program logic from its scheduling*. In particular, as a contribution, we:

- propose a novel programming model and language, called *Functional Reactive Approach to Self-organisation Programming (FRASP)*;
- provide an open-source implementation as a Scala domain-specific language (DSL)[1], leveraging the functional reactive library Sodium and inspiration from the ScaFi aggregate programming DSL [19], [20];
- experimentally evaluate the benefits of reactivity and resource usage through an open-source, permanently available artefact with reproducible simulations[2].

The result is a functional reactive self-organisation programming model, relying on functional composition of behaviours,

This work has been supported by the Italian PRIN Project COMMON-WEARS (2020HCWWLP) and the EU/MUR FSE REACT-EU PON R&I 2014-2020.

[1]https://github.com/cric96/distributed-frp
[2]https://github.com/AggregateComputing/experiment-2023-acsos-distributed-frp

and whereby each sub-expression is amenable to independent scheduling: overall, we maintain the same expressiveness and benefits of aggregate programming [21], [22] while enabling significant improvements in terms of scheduling controllability, flexibility in the sensing/actuation model, and execution efficiency.

The rest of the article is organised as follows. Section II covers background on self-organisation programming and the functional reactive paradigm. Section III presents the FRASP programming model. Section IV details the implementation. Section V describes the experimental setup and results. Section VI draws conclusions and future research work.

II. Background and Motivation

In this section, we review approaches for self-organisation engineering (Section II-A), and set the goal of combining the benefits of reactive approaches with those of compositional macroprogramming models like aggregate computing. Then, we provide background on functional reactive programming (FRP) (Section II-B), the paradigm we choose for our programming model, for its benefits in declarativity and automatic, configurable management of change.

A. Self-organisation Engineering Approaches

The first distinction in approaches for self-organisation engineering is between automatic and manual approaches. In the former category, the control logic of the agents is learned, synthesised, or evolved automatically—as e.g., in multi-agent reinforcement learning [10], Collective Intelligence [23], [24], and evolutionary swarm robotics [25]. The latter category, instead, involves explicit engineering of the activity by a developer, e.g., by reusing patterns of behaviour [26], by directly coding the individual logic of each type of device, or by using mechanisms provided by a so-called *macroprogramming language* [13], [14]. The rest of the section covers prominent approaches in this category (i.e. programming models)—distinguishing between reactive and round-based ones.

Among reactive approaches is *Tuples On The Air (TOTA)* [27], a programming model for decentralised peer-to-peer networks of mobile nodes or agents. It uses *tuples* to represent context information and mediate interactions between agents. In particular, tuples are *reactive*: they are associated with propagation rules that describe how tuples should be propagated to neighbours (hop-by-hop) in a network and how the content of tuples should change during propagation or in reaction to environmental events. The agents behave and coordinate through operations on tuples (e.g., insertion, read, removal, waiting) or by subscribing to tuple-related events. Other reactive approaches exist, such as the *Higher-Order Chemical Language (HOCL)* [28], but they feature quite a large abstraction gap.

On the other side, there are programming models based on a *round-based* execution model whereby each device repeatedly performs a complete evaluation of its control program (e.g., wrapped in a loop or scheduled in a periodic, time-driven fashion). Aggregate computing [21], [22] is one such approach, where each round is atomically composed of *sense–compute–interact* steps, and a language is used to express the *compute* step conveniently. Indeed, it emerged as a prominent approach for programming self-organisation [22], with the benefits of formality, abstraction, compositionality, and pragmatism. Formality stems from building the approach over *field calculi* [22] with well-defined language semantics. Abstraction comes from the declarativity of the *functional* programming model, promoting different scheduling strategies [29], [30] and deployments [18], possibly applied automatically to different sub-activities concurrently running. Compositionality comes from adopting the functional paradigm and the *field abstraction* [22], [31] (a field is a map from a domain of devices to computational values—e.g., a field of temperatures, or a field of velocity vectors). Finally, pragmatism is supported by the language design and its separation of concerns, which enables modular DSLs and toolkits [19].

Though conceptually simple, the round-based models could be more efficient, because they fully re-evaluate the context and the whole program without tracking change. Though it might be acceptable for predictable patterns of environmental change, this becomes largely suboptimal for highly variable dynamics. Indeed, the round-based approach seems to be a legacy of imperative languages or solutions featuring limited compositionality. Instead, more compositional languages like, e.g., aggregate computing languages [22], [32], [33] and Buzz [34], allow building complex self-organising behaviour by composing *blocks* of simpler self-organising behaviours—see Section III-C for examples. Therefore, each individual block of behaviour is *potentially independent of others (i.e., independently schedulable)*, with data dependencies arising from each composition. A reactive extension of aggregate computing has been proposed in [30], based on manually specifying dependencies and reactive policies (with configurable triggers) among different aggregate computations. Instead, a more practical approach could be decoupling programs from the specification of reactive policies and letting them only define the data dependencies between program portions. The most suitable programming approach for this is the *functional reactive programming* paradigm [35], briefly introduced in the following.

B. Functional Reactive Programming

Reactive programming [35] is a paradigm suitable for developing event-driven applications, leveraging abstractions to express (relationships between) time-varying values and automatically handle the propagation of change (cf. the paradigmatic example of spreadsheets [36]). Reactive programming is often combined with the functional programming paradigm [35], [36] in the so-called FRP.

FRP builds on few abstractions and various combinators [37]. Conceptually, FRP considers *continuous time*, $Time = \{t \in \mathbb{R} \mid t \geq 0\}$. Time-varying values are called *cells* and may be conceptually modelled by generic functions of type $Cell\ a : Time \rightarrow a$. Then, *streams* are discrete-time values and may be modelled by generic functions of type $Stream\ a : [Time] \rightarrow [a]$ (where notation $[X]$ indicates sequences of Xs), namely, mapping a sequence of (increasing) sample times to a sequence of corresponding values. While cells model state, streams model state changes (or events). Then, FRP libraries provide functions (combinators) for transforming signals to signals, streams to streams, signals to streams, and streams to signals. An example of such a library is Sodium [36], the Java library for FRP which we leveraged to implement FRASP as described in Section IV.

III. The FRASP Programming Model

This section presents the FRASP programming model from a user perspective. First, we explain the system and execution model (Section III-A); then, we present the language abstractions and primitives (Section III-B); finally, we show how paradigmatic examples of self-organisation can be expressed in FRASP (Section III-C).

A. System Model and (Reactive) Execution Model

We shall program the self-organising behaviour of a *system* of *devices*. A *device* has a *unique identifier (ID)*, as well as *sensors* and *actuators* that allow it to perceive and act upon its surrounding *environment*. A device can interact with other devices (its *neighbours*, forming a dynamic set) by exchanging messages asynchronously. A device has an associated *program* that defines its behaviour.

The program that a device runs can be defined compositionally, and it generally computes over its *local context*, which consists of (i) a sampling of its sensor values and (ii) the data provided by neighbours through messages. Without loss of generality, we assume that all the data provided by a neighbour is consolidated into a single object (i.e., *one object per neighbour*) and that neighbour data may *expire* (i.e., it may be set to be valid for a configurable, limited lifetime). The execution of a program (computation) provides an *output* object and defines what data has to be packed into a message, also called an *export*, to be sent to all the neighbours. The export can be thought of as an associative map of keys and values, where keys denote different sub-computations and values the data relevant to those sub-computations.

In general, the *scheduling* of a program execution is asynchronous w.r.t. other devices and may be periodic (time-triggered) or reactive. In this work, we consider reactive scheduling and compare it with the periodic scheduling of earlier research. In reactive settings, a program may need to be re-executed any time an input changes, i.e.:

- sensor data (e.g., the temperature sensor perceives a different temperature);
- neighbour data (e.g., a device is no longer a neighbour, a message has expired, or a neighbour provides a more recent message that supersedes previous data).

A re-evaluation of the program may produce a different output and export. Furthermore, in this work, we take this reactivity scheduling of programs a step further by allowing individual *expressions* (i.e., portions of programs) to be re-evaluated when their context and inputs (e.g., dependencies on other expressions) change—thanks to the FRP approach. This idea will be shown in Example 2 and Figure 1.

Notice that the model is logical and may be implemented using different approaches and optimisations—e.g., a device may send a heartbeat to notify that it is still a neighbour and its data has not changed, it may send a message with only data that has changed, and so on. Also, inbound and outbound reactivity can be regulated by throttling, i.e., by accumulating a certain amount of (change in) inputs (before re-evaluating the program or parts of it) and accumulating a certain amount of (change in) outputs/exports (before executing actuations and/or sending the export to neighbours).

B. Programming Abstractions and Primitives

We adopt a *macroprogramming* approach [13], where a single program is used to express the collective behaviour of the entire system of devices. Specifically, the self-organising collective behaviour will *emerge* from (i) the local execution of the program by all the devices making up the system, (ii) the distributed execution (implementing the message passing), and (iii) the environment dynamics (which will affect neighbourhoods and the data perceived by sensors).

Since FRASP is implemented as a DSL internal (or embedded) in Scala, Scala types and features (e.g., functions) can be reused in FRASP programs.

1) Datatypes: According to the FRP paradigm, we would like to express a self-organising collective computation as a graph of reactive sub-computations. We call each sub-computation a *flow* and represent it programmatically through type `Flow[T]`[3], where `T` is the type of the output of the wrapped computation. A `Flow` is essentially a function that takes a `Context` and returns a *cell* of `Export`s, possibly depending on the exports of other `Flow`s, recursively—see Section IV-C for details. With abuse of terminology, we will refer to a flow as its output cell, i.e., as a time-varying value.

2) Local values: The simplest constructs of the language are local and atomic (i.e., that do not depend on other flows or neighbours).

- `constant(e)` returns a constant flow that always evaluates to the argument that has been passed;
- `sensor(name)` returns the flow of values produced by the sensor with the given `name`;
- `mid()`, as a shortcut to `sensor("mid")`, returns the constant flow of the device ID.

3) Choice: A `mux(c){t}{e}` expression returns a flow with the same output of flow `t` when the Boolean flow `c` is true and the output of flow `e` when `c` is false. E.g., code

```
mux(sensor("temperature") > THRESHOLD)
    { constant("hot") } { constant("normal") }
```

will yield the string `"hot"` in the devices where the local temperature sensor yields a value below the given threshold and the string `"normal"` otherwise.

[3]Notation: we highlight types in `brown`, primitives in `red`, derived/library constructs in `purple`, and Scala (host language) keywords in **`blue`**.

4) Interaction with neighbours: Communication with neighbours is handled *in both directions at once* through a single construct, `nbr(f)`, which takes a flow `f` as parameter. The local output of `f` will be automatically sent to neighbours. Instead, the output of the whole `nbr(f)` expression is an object `NeighborField[T]` collecting the values of `f` computed by all the neighbours. For instance

```
nbr(mid()) // or nbr(mid()).withoutSelf to exclude "self"
```

returns, in any device, the IDs of all its neighbours (including the device itself).

Sensors providing a value for each neighbour have dedicated syntax. They can be queried through construct `nbrSensor(name)`. For instance, the built-in function `nbrRange`, defined as follows:

```
def nbrRange(): Flow[NeighborField[Double]] =
  nbrSensor("nbrRange")
```

provides the neighbouring field of (estimated) distances to neighbours (how such a sensor works is an implementation detail—e.g., it may use GPS traces or WiFi signal strength).

5) Branching: An expression `branch(c){t}{e}` evaluates and returns the value of expression `t` (resp. `e`) when `c` evaluates to `true` (resp. `false`). This enables a form of distributed branching, where devices that happen to execute `t` will not interact with those that executed `e` (and vice versa)—unlike `mux` in which a device "contributes" to both `t` and `e`. E.g., in a system split into red and blue devices, the expression:

```
branch(sensor("color") == "red"){
  nbr(constant(1)).sum // red nodes run this
} {
  nbr(constant(1)).sum // blue nodes run this
}
```

will yield in any device the number of neighbours *of the same kind*, neighbours that run the other sub-computation (despite those being the same) will not be considered. This concept is called *alignment* and is well-discussed, e.g., in [20]. Notice that, upon a change in the value sensed by `color`, a device may dynamically switch branch and hence sub-computation domain and "aligned" neighbour set.

6) Lifting: Lifting enables flow combination: i.e., `lift(f1,f2,...,fN){g}` yields a flow obtained by applying `g(o1,o2,...,oN)` where `oi` is the output of flow `fi`. Lifting can also be applied on flows of `NeighborField`s, in which case the output is a flow of a `NeighborField` whose values are combined from the input `NeighborField`s neighbour-wise (runtime checks avoid combining flows with different domains). E.g.,

```
lift(nbr(mid(),nbrRange()){(nbrId,nbrDist) =>
  s"${nbrId}␣is␣at␣distance␣${nbrDist}␣from␣me"}
```

yields locally to a device one string per neighbour reporting its ID and distance.

7) Looping (state evolution): Construct `loop(init,ft)` evolves a piece of state (initially, `init`) by applying function `ft` mapping the previous state's flow to the next state's flow. For instance, the expression:

```
loop(0)(v => v + 1)
```

represents a computation counting from 0 onwards (ignoring overflow). How frequently does this counting progress? It depends on the implementation of `Context`, which provides a default throttling period. A different `loop` implementation may also accept a stream explicitly dictating the pace of the stateful computation (e.g., evolving state each time a button is pressed).

C. Paradigmatic Examples: Self-Healing Gradient & Channel

In this section, we cover two fundamental self-organising behaviours that will be exercised in the evaluation in Section V. Firstly, we introduce the *gradient* [38], [39], a versatile self-organisation pattern that supports a wide range of more complex self-organising behaviours.

Example 1 (Self-Healing Gradient)**.** A *(self-healing) gradient* is a distributed behaviour that *self-stabilises* [40], in each device of the distributed system, to a value denoting its minimum distance from the closest *source* node (e.g., computed by summing the neighbour-to-neighbour distances along the shortest path to the source), adapting to changes in the source set and distances. By following the neighbours of maximum decrease (resp. increase) of the gradient value, i.e., by descending (resp. ascending) the gradient, it is possible to implement efficient hop-by-hop information flows [11], e.g., useful for data propagation and collection. Figure 4a provides a representation of a gradient.

Multiple gradient computation algorithms exist [39]. A basic algorithm can be implemented in FRASP as follows.

```
def gradient(source: Flow[Boolean]): Flow[Double] =
  loop(Double.PositiveInfinity) { g => {
    mux(source) {
      constant(0.0)
    } {
      lift(nbrRange(), nbr(g)){_ + _}
        .withoutSelf
        .min
    }
  }
```

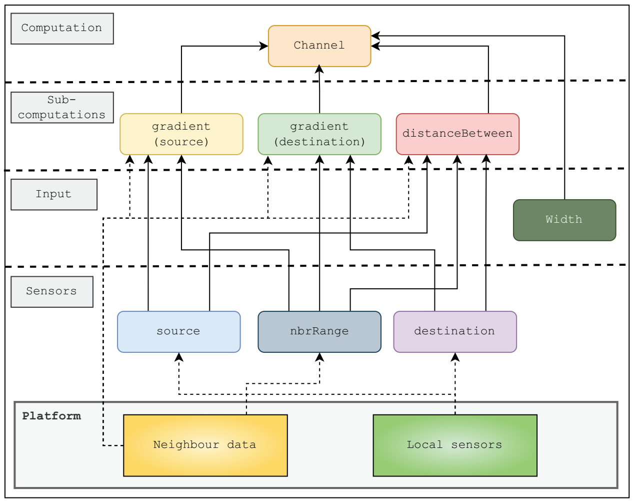
The function takes the Boolean `source` flow as input, denoting whether the executing node is the source of the gradient or not. The external `loop` is used to progressively evolve the current gradient value `g` starting from an infinite value (as, initially, we do not know whether a source is reachable). Internally to the loop, we use `mux` to select one of two values: if the node is a source (i.e., `source` is true), then its gradient value is `0` (base case); otherwise, the gradient should be the minimum value among the neighbours' gradient values augmented by the distance (`nbrRange`) from that very neighbour. Construct `lift` is used to combine (using the sum, cf. _+_) the two flows `nbrRange` (distances to neighbours) and `nbr(g)` (neighbours' gradient values).

The following example showcases the *compositionality* of the programming model, namely the possibility of combining multiple self-organising behaviours to build a more complex self-organising behaviour.

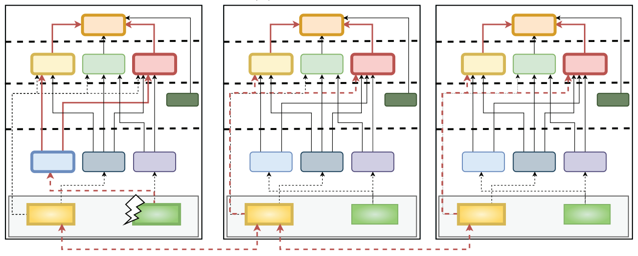
Example 2 (Self-Healing Channel)**.** A *channel* computation aims to build the shortest path from a *source* node to a *destination* node. Each node has to yield a Boolean value that should eventually be `true` if the node belongs to the shortest path and `false` otherwise. The basic idea for its implementation is to exploit the triangular inequality property, i.e., that the "distance to the source" plus the "distance to the destination" is less than or equal to the "distance between the source and the destination". Therefore, an implementation in FRASP is as follows.

```
 1 def broadcast[T](
 2   source: Flow[Boolean], value: Flow[T]
 3 ): Flow[T] =
 4   val broadcastResult = loop[(Double, Option[T])](
 5     (Double.PositiveInfinity, None)
 6   ) { d =>
 7     val x = value.map(0.0 -> Some(_))
 8     val y =
 9     mux(source) { value.map(0.0 -> Some(_)) } {
10       val n = nbr(d)
11       val distances = n.mapTwice(_._1)
12       val values = n.mapTwice(_._2)
13       val field = lift(distances, nbrRange(), values) {
14         (ds, ra, va) => (ds + ra) -> va
15       }
16       field.withoutSelf
17         .map(_.values.minByOption(_._1)
18         .getOrElse((Double.PositiveInfinity,None)))
19     }
20   }
21   lift(broadcastResult, value){_._2.getOrElse(_)}
22
23 def distanceBetween(
24   source: Flow[Boolean], destination: Flow[Boolean]
25 ): Flow[Double] =
26   broadcast(source, gradient(destination))
27
28 def channel(
29   source: Flow[Boolean],
30   destination: Flow[Boolean],
31   width: Double,
32 ): Flow[Boolean] = lift(
33   gradient(source),
34   gradient(destination),
35   distanceBetween(source, destination)
36 ){
37   (distSource, distDest, distBetween) =>
38     distSource + distDest <= distBetween + width
39 }
```

This self-organising data structure can be implemented by leveraging two `gradient`s (one from the source and one from the destination—cf. Lines 33 and 34) and a `broadcast(v,s)` (which is a way to propagate a value `v` hop-by-hop from a source `s` outwards along the minimum paths of its gradient—indeed, a structure similar to the `gradient` implementation in Example 1) that supports the computation of `distanceBetween`. The `channel` depends on these three flows: i.e., the expression at Line 38 will be re-evaluated only upon a change of the output of (one of) these three sub-computations. For a graphical view of the local and distributed dependency graph, see Figure 1: the key idea is that, e.g., a local change in sensor `source` will not cause a re-computation of `gradient(destination)`.



(a) Node view.



(b) Distributed view (with neighbour dependencies).

Fig. 1: The reactive dataflow graph corresponding to Example 2. Figure 1a provides the local view of the computation for a single node (where the layers denote different semantic kinds of dependencies), whereas Figure 1b shows the *distributed* dependency graph. The arrows denote dependencies. The dashed arrows denote dependencies based on platform-level scheduling and node interaction—e.g., a red block depends on changes corresponding to neighbours' red blocks and communicated via message passing.

IV. Implementation

This section briefly provides the implementation goals (Section IV-A), architectural design (Section IV-B), and implementation details (Section IV-C) of FRASP. Even though a complete description of the implementation is beyond the aims of this paper, this section is meant to illustrate that the prototype is technically sound, that we followed modern software engineering practices, and to provide general guidance for understanding the code organisation of the provided artefact (see Footnote 1).

A. Goals

The high-level goal of this work is to provide a programming model that is expressive enough to allow developers to declaratively describe *self-organising* collective computations while decoupling and providing fine-grained control over their scheduling details. This vision can be summarised with the

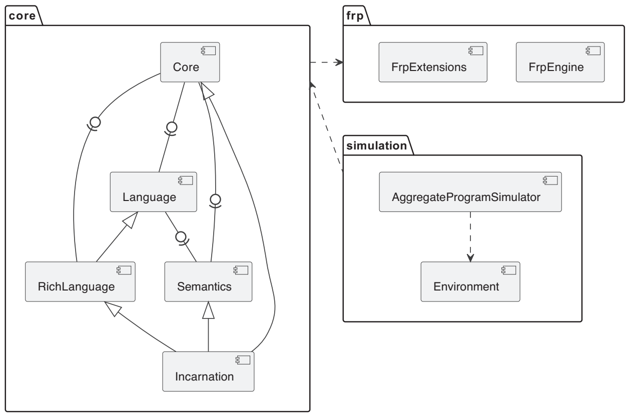


Fig. 2: Architecture of FRASP.

term *functional reactive self-organisation*. Considering the system model introduced in Section III-A, there are three main objectives to be pursued in order to accomplish the goal:

- *Re-compute only upon* relevant *changes in the context*: computations should occur reactively only when relevant changes are observed from the environment (i.e., sensing and neighbour data).
- *Avoid re-evaluation of unaffected sub-computations*: if a portion of the computation depends on data that did not change, it should not be re-evaluated.
- *Interact only upon relevant changes*: each device should avoid broadcasting an export that did not change since the last one, with the direct consequence that no further message exchange should be required if a computation reaches a stable configuration.

B. Architecture

The architecture of the prototype is shown in Figure 2. The design is organised into three packages: `core`, which includes basic type definitions (`Core`) as well as the components for the DSL (`Language` for primitives and `RichLanguage` for other built-ins) and its "virtual machine" (`Semantics`), overall captured by an `Incarnation`; `frp`, which provides an interface to the FRP engine (`FrpEngine`), possibly also decoupling from the specific FRP library adopted, as well as extensions (`FrpExtensions`) useful for the definition of FRASP constructs; and `simulation`, which provides basic simulation support (for more advanced support, we also integrated FRASP into Alchemist [41]—see Section V).

C. Implementation details

FRASP has been implemented in Scala, using Sodium as FRP library [36]. Scala is well-known for its suitability as a host for embedded DSLs [42], and for aggregate computing embeddings as well [20]. The design of the FRASP DSL is detailed in Figure 3.

Following the system/execution model described in Section III-A, we model the input and output of a (sub-)program through an interface `Context`, providing access to local

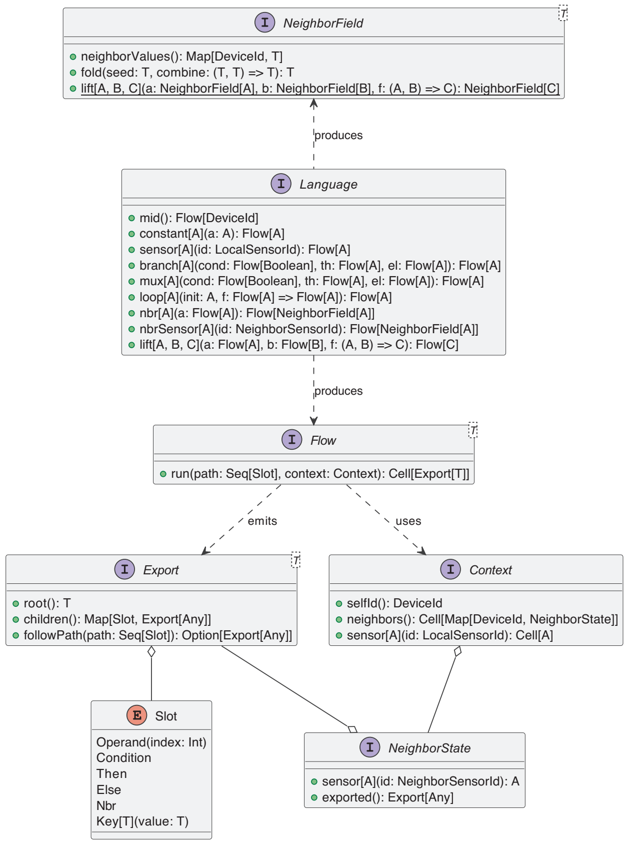
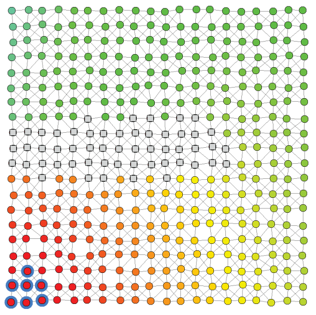

I NeighborField
T
neighborValues(): Map[DeviceId, T]
fold(seed: T, combine: (T, T) => T): T
lift[A, B, C](a: NeighborField[A], b: NeighborField[B], f: (A, B) => C): NeighborField[C]
produces
I Language
mid(): Flow[DeviceId]
constant[A](a: A): Flow[A]
sensor[A](id: LocalSensorId): Flow[A]
branch[A](cond: Flow[Boolean], th: Flow[A], el: Flow[A]): Flow[A]
mux[A](cond: Flow[Boolean], th: Flow[A], el: Flow[A]): Flow[A]
loop[A](init: A, f: Flow[A] => Flow[A]): Flow[A]
nbr[A](a: Flow[A]): Flow[NeighborField[A]]
nbrSensor[A](id: NeighborSensorId): Flow[NeighborField[A]]
lift[A, B, C](a: Flow[A], b: Flow[B], f: (A, B) => C): Flow[C]
produces
I Flow
T
run(path: Seq[Slot], context: Context): Cell[Export[T]]
emits
uses
I Export
T
root(): T
children(): Map[Slot, Export[Any]]
followPath(path: Seq[Slot]): Option[Export[Any]]
I Context
selfId(): DeviceId
neighbors(): Cell[Map[DeviceId, NeighborState]]
sensor[A](id: LocalSensorId): Cell[A]
E Slot
Operand(index: Int)
Condition
Then
Else
Nbr
Key[T](value: T)
I NeighborState
sensor[A](id: NeighborSensorId): A
exported(): Export[Any]

Fig. 3: Design of FRASP DSL.

sensor data and neighbour data; and an interface `Export`, capturing outputs and data that must be shared with neighbours. In particular, an `Export` is modelled as a *tree* where each node is a `Slot` (corresponding to a particular language construct) with an associated value, and can be located through a *path* of slots—e.g., `S1/S2/S3` identifies a node in the export tree, where `S1` depends on `S2` which depends in turn on `S3` (so, a change in the output `S3` will cause the expression corresponding to `S2` to re-evaluate, and possibly `S1` in turn).

`Flow` is the type of a reactive (sub-)computation, which takes a `Context` (providing its inputs), a `Seq[Slot]` as path (indicating its position in the export tree), and returns `Cell` (i.e. a time-varying value—cf. Section II-B) of `Export`. Each `Language` construct returns a `Flow`: therefore, the constructs do *not* immediately run upon evaluation, but rather an executable, reactive object denoting a computation graph whose nodes will execute as a response to change (cf. Figure 1). Access to neighbour-related data is mediated by a `NeighborField` abstraction, which is the same provided by constructs supporting interaction with neighbours, i.e., `nbr` and `nbrSensor` (cf. Section III-B).


(a) Scenario G: gradient.

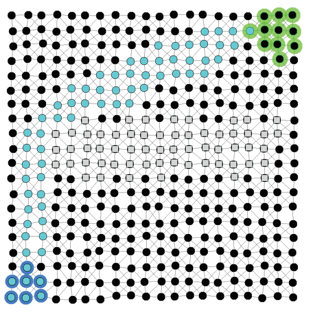

(b) Scenario C: channel.

Fig. 4: Evaluation scenarios. The output of the program being evaluated is depicted in the inner circle. On the left, the hue varies depending on the gradient output, with redder colours indicating lower values. On the right, the channel between the source nodes (blue shadow) and destination (green shadow) is indicated in cyan. The grey nodes denote obstacles.

V. Evaluation

To evaluate the proposed approach, we prepared several publicly available and reproducible simulations (see Footnote 2) using the *Alchemist* simulator for large-scale pervasive systems[4] [41]. In particular, we released FRASP[5] and integrated it into Alchemist. The choice of Alchemist allowed us to leverage the already integrated *ScaFi* aggregate programming language [19] as a reference for round-based computations (cf. Section II-A).

A. Goals

The simulations are designed to evaluate the following:

G.1) *Correctness*: the reactive and round-based versions of the same algorithm should ultimately produce the same correct collective results.

G.2) *Efficiency*: The efficiency is measured in terms of:
   a) *messages*: the number of messages exchanged between devices;
   b) *time*: the time required to reach a stable output;
   c) *computation*: the number of sub-computation steps performed by each device.

   In particular, we expect the reactive version of the algorithm to be *more efficient* than the round-based one.

G.3) *Reactivity*: the programs should react to various sources of change (e.g., network topology, sensor data, dependent computations—cf. Section III-A), avoiding re-evaluations when inputs do not change.

B. Experimental Setup

1) Common setup and parameters: The simulated system consists of 400 devices placed on a $100m \times 100m$ square and forming a slightly irregular grid (nodes' positions are generated for a regular grid and then randomly deviated). Each node has a mean distance to its neighbours of 5 meters and a communication radius of 7.5 meters. Messages sent by the nodes may be subject to a communication delay regulated by the parameter τ, which describes an exponential probabilistic function. Each simulation is characterised by the mode of execution of the aggregate program, which can be (i) *purely reactive*, (ii) *reactive with throttling*, or (iii) *round-based*. The pure reactive policy is evaluated for every new message received from any neighbour. This policy is expected to converge quickly at the price of a significant overhead in the number of exchanged messages. The "reactive with throttling" policy is parametrised on the throttling frequency γ (i.e., the inverse of the period in which all events received in this interval are accumulated before emission). Compared to the purely reactive policy, throttling is expected to reduce message overhead at the expense of convergence time. Finally, the round-based policy is driven by a γ parameter describing how often each device will wake up and compute the round.

2) Scenarios: In the above setup, the two programs introduced in Section III, i.e., the self-healing gradient (*scenario G*—cf. Figure 4a) and channel (*scenario C*—cf. Figure 4b), are executed for 300 simulated time units. The former represents a minimally complex self-organising behaviour, enabling evaluation of basic dynamics, whereas the latter is representative of larger behaviours that can be defined as compositions of simpler ones, hence providing insights about what could happen when multiple reactive computations are combined.

For *scenario G*, a group of nodes (i.e., a $20m \times 20m$ area at the bottom left) is marked as gradient source. Also, a set of nodes marked as obstacles is positioned at the centre in a $8m \times 2m$ area. To verify that the gradient can adapt to changes and assess the effects of continuous and frequent changes in the system, at simulated time $t = 150$ a cluster of nodes migrates from the lower left-hand side to the lower right-hand side of the area.

For *scenario C*, a set of nodes denoting the channel's destination are placed in the upper right in a $20m \times 20m$ area. Here, to verify reactivity to change, we switch the set of destination nodes at $t = 200$ from the upper-right corner to the upper-left corner. This injected change allows us to observe both the channel computation's reactivity and the sub-computations' evaluation. In fact, by only modifying the destination area, the gradient starting from the source (which is a sub-computation of the channel computation) should not be re-evaluated (as it would not change its collective output).

3) Metrics: The metrics extracted for this study are:

- *Total cumulative number of messages exchanged (up to time T)* – `#messages`: this is used to evaluate the communication overhead/efficiency (cf. goal G.2) and to inspect the communication dynamics of the reactive solution. It is computed as the sum of the number of messages sent by each node up to time T.
- *Average output value of the system* – `output (mean)`: for each time instant t, we extract the average value of the computational field produced by the system. This value

[4]https://alchemistsimulator.github.io/

[5]https://central.sonatype.com/artifact/io.github.cric96/distributed-frp_3/0.1.3

will allow us to assess both correctness (cf. goal G.1) and time efficiency (when the programs converge to a certain stable value—cf. goal G.2). Also, this indicates the responsiveness of the system, as the output should vary with the introduction of changes in the system.

- *Total number of executions for program sub-computations* – `#evaluations`: this value was extracted in the channel program only (since it comprises multiple subprograms). For each sub-program, we calculated the number of times it has been evaluated up to time T as the sum of the number of evaluations of each node.
  This metric is useful for assessing goals G.2c and G.3.

4) Baselines: To establish baselines, we implemented round-based solution programs for the self-healing gradient (scenario G) and channel (scenario C) in ScaFi—see [19]. The execution for the round-based solution is configured with an evaluation frequency of 1Hz: any device will evaluate the entire ScaFi program every second and then broadcast the resulting export to its neighbourhood (even if it did not change from the previous execution).

C. Results and Discussion

In this section, we will present the results obtained from the simulations, highlighting how the proposed model satisfies the goals elicited in Section V-A. We run a total of 768 simulations by running 64 randomised instances (varying the random seed and the position of the nodes in the environment) for each one of the 12 *simulation configurations* obtained by a different combination of *(i)* execution mode, *(ii)* throttling period, and *(iii)* scenario program. Our results are presented in Figure 5 and Figure 6.

1) Correctness (goal G.1): Consider Figure 5b and Figure 5d: we observe that in all cases, especially in the reactive and throttling executions, the program output converges to the same collective result. Pure reactive policies have less noise (as variations are reduced) but tend to converge to the same result on average.

2) Efficiency (goal G.2): From Figure 5, we can see how the reactive computation is more efficient than the round-based one (cf. goal G.2) in the sense of communication/computation efficiency (Figures 5a and 5c) and time efficiency (Figure 5d). Moreover, as expected, throttling can further improve the efficiency of the reactive solution.

Indeed, starting with the messages metric, we observe that the number of messages exchanged in pure reactive policies to reach a stable output is greater than the throttled counterpart. This result is expected: in the purely reactive model each message different from a previous one causes a program re-evaluation and subsequent message sending. Analysing the policies with throttling, we notice that the higher the throttling period, the lower the number of messages sent, and in particular, the more the policy tends towards the round-based one. In the gradient scenario, for instance, the throttle mode achieves convergence by sending six times fewer messages than the round-based one. Despite consuming more, purely reactive policies are still more efficient than round-based ones, sending approximately 40% less messages. However, it is also noteworthy that, in the case of continuous variations, such as the migration of nodes in the gradient scenario, the number of messages grows linearly in both reactive and round-based modes. Efficiency could be improved by approximating the output value not to send every message for every update, i.e., to regulate reactivity according to some threshold for "significant change".

Moving on to converge time, we immediately note that reactive policies are the most efficient because they expand changes throughout the system as soon as they occur, avoiding delays due to waiting for a new evaluation round. However, in the case of frequent environmental change, one may have a higher message consumption than round-based policies, leading to higher energy consumption.

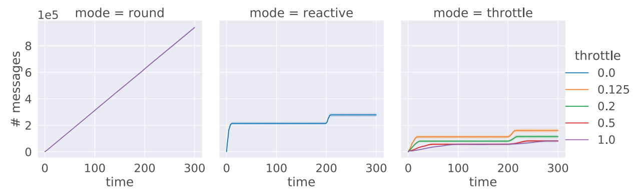
Observing throttling policies, we note that the higher the throttling period, the higher the convergence time, as the system will take longer to react to changes. In the worst case (i.e., when the throttling period is equal to the evaluation time of round-based policy), the convergence time is approximately the same. However, convergence time shortens with the decrease in the throttling period, getting performance close to the purely reactive policy but with fewer messages exchanged. Thus, *the proposed model balances communication cost (the number of messages exchanged) and performance (time required to converge to the expected output), depending on the application's needs*. Moreover, the relationship between the throttling period and the number of messages sent is not linear (unlike the convergence time). In fact, halving the period (0.5 seconds instead of 1 second) also halves the convergence time, yet the number of messages exchanged is approximately the same.

Concerning reactivity (goal G.3), we notice how the reactive solutions effectively "follow" environmental changes. This can be observed from Figure 5a and Figure 5c: when there is no change to react to, no program evaluation is performed, resulting in no new messages being sent (cf. the intervals where the message count does not increase). Moreover, as the environmental conditions vary, the program is re-evaluated, as observed at the moment of variation (at $t = 150$ in the case of the gradient and at $t = 200$ in the case of the channel).

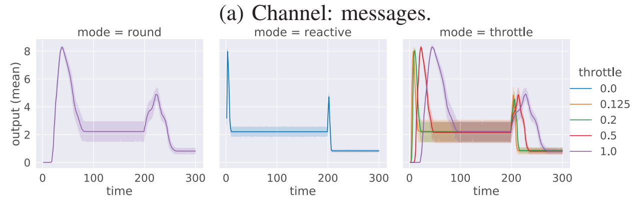
Finally, we show how the model promotes *fine-grained reactivity*. Figure 6 shows how FRASP enables a program to re-compute required sub-computations *only*. Indeed, at $t = 200$, only the channel's target changes, and thus the source gradient is not re-computed (it would be pointless since its inputs did not change). Also, this execution occurs only where needed, resulting in a "wave" of re-computations from the source to the destination (cf. the larger red circles in Figure 6).

VI. Conclusions and Future Work

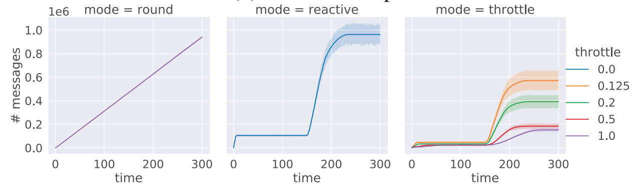
This paper proposes FRASP, a *functional reactive macro-programming model for expressing self-organising behaviour*. The language is designed by taking inspiration from the aggregate programming approach, which is based on a proactive execution model, and reactive approaches to self-organisation



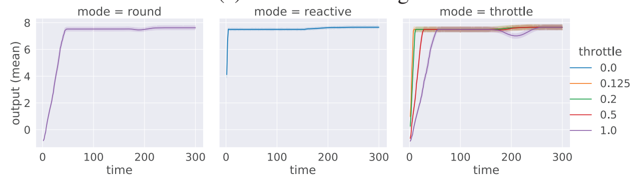
(a) Channel: messages.



(b) Channel: output.



(c) Gradient: messages.



(d) Gradient: output.

Fig. 5: Simulation data. The output value (lines 2 and 4) from the round-based solution (left column) can be used as a reference to verify the correctness of the computation and the performance of the reactive solutions (middle and right columns). The number of messages (lines 1 and 3), instead, provides an indication of the communication overhead. We observe that purely reactive (middle column) and throttled (right column) policies convergence to correct values faster than the round-based version; also, their communication overhead does not grow with time, but depends on changes.

like TOTA, and adopting FRP as a reference paradigm for its design and implementation. As experimentally verified, FRASP allows *tunable, fine-grained reactivity*, enabling increased communication and time efficiency w.r.t. proactive models. Indeed, in FRASP, a distributed self-organising computation turns into a distributed computation dependency graph, where distinct sub-computations may execute independently depending on whether their context has changed.

In future work, we would like to formalise a core calculus capturing behaviour and properties of FRASP (similarly to field calculi and related formal languages [20], [22], [33]), and explore how to support flexibility in actuation models (cf. swarm robotics). Also, it would be interesting to study the deployment of FRASP applications, e.g. by considering how reactive dynamics and consumption profiles may interact with application partitioning strategies like the pulverisation model [18] and deployment options.

References

[1] H. V. D. Parunak and S. A. Brueckner, "Software engineering for self-organizing systems," *Knowl. Eng. Rev.*, vol. 30, no. 4, pp. 419–434, 2015. doi: 10.1017/S0269888915000089

[2] C. Gershenson, *Design and control of self-organizing systems*. CopIt Arxives, 2007.

[3] V. K. Singh, G. Singh, and S. Pande, "Emergence, self-organization and collective intelligence - modeling the dynamics of complex collectives in social and organizational settings," in *UKSim*. IEEE, 2013. doi: 10.1109/UKSim.2013.77 pp. 182–189.

[4] R. D. Nicola, S. Jähnichen, and M. Wirsing, "Rigorous engineering of collective adaptive systems: special section," *Int. J. Softw. Tools Technol. Transf.*, vol. 22, no. 4, pp. 389–397, 2020. doi: 10.1007/s10009-020-00565-0

[5] M. Prokopenko, "Guided self-organization," 2009.

[6] H. Schmeck, C. Müller-Schloer, E. Cakar, M. Mnif, and U. Richter, "Adaptivity and self-organization in organic computing systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 5, no. 3, pp. 10:1–10:32, 2010. doi: 10.1145/1837909.1837911

[7] F. Varenne, P. Chaigneau, J. Petitot, and R. Doursat, "Programming the emergence in morphogenetically architected complex systems," *Acta Biotheoretica*, vol. 63, no. 3, pp. 295–308, 2015. doi: 10.1007/s10441-015-9262-z

[8] K. Giammarco, "Practical modeling concepts for engineering emergence in systems of systems," in *SoSE*. IEEE, 2017. doi: 10.1109/SYSOSE.2017.7994977 pp. 1–6.

[9] G. Martius and J. M. Herrmann, "Variants of guided self-organization for robot control," *Theory Biosci.*, vol. 131, no. 3, pp. 129–137, 2012. doi: 10.1007/s12064-011-0141-0

[10] K. Zhang, Z. Yang, and T. Basar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *CoRR*, vol. abs/1911.10635, 2019.

[11] T. D. Wolf and T. Holvoet, "Designing self-organising emergent systems based on information flows and feedback-loops," in *SASO*. IEEE Computer Society, 2007. doi: 10.1109/SASO.2007.16 pp. 295–298.

[12] G. Aguzzi, R. Casadei, and M. Viroli, "Towards reinforcement learning-based aggregate computing," in *COORDINATION*, ser. LNCS, vol. 13271. Springer, 2022. doi: 10.1007/978-3-031-08143-9_5 pp. 72–91.

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

[14] I. G. S. Júnior, T. S. Santana, R. F. Bulcão-Neto, and B. Porter, "The state of the art of macroprogramming in IoT: An update," *J. Internet Serv. Appl.*, vol. 13, no. 1, pp. 54–65, 2022. doi: 10.5753/jisa.2022.2372

[15] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intell*, vol. 7, no. 1, pp. 1–41, 2013. doi: 10.1007/s11721-012-0075-2

[16] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, 2011. doi: 10.1145/1922649.1922656

[17] J. Noor, H. Tseng, L. Garcia, and M. B. Srivastava, "DDFlow: visualized declarative programming for heterogeneous iot networks," in *IoTDI*. ACM, 2019. doi: 10.1145/3302505.3310079 pp. 172–177.

[18] R. Casadei, D. Pianini, A. Placuzzi, M. Viroli, and D. Weyns, "Pulverization in Cyber-Physical Systems: Engineering the Self-Organizing Logic Separated from Deployment," *Future Internet*, vol. 203(12), Nov. 2020. doi: 10.3390/fi12110203

[19] R. Casadei, M. Viroli, G. Aguzzi, and D. Pianini, "ScaFi: A Scala DSL and toolkit for aggregate programming," *SoftwareX*, vol. 20, p. 101248, 2022. doi: 10.1016/j.softx.2022.101248

[20] G. Audrito, R. Casadei, F. Damiani, and M. Viroli, "Computation against a neighbour: Addressing large-scale distribution and adaptivity with functional programming and Scala," *Log. Methods Comput. Sci.*, vol. 19, no. 1, 2023. doi: 10.46298/lmcs-19(1:6)2023

[21] 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

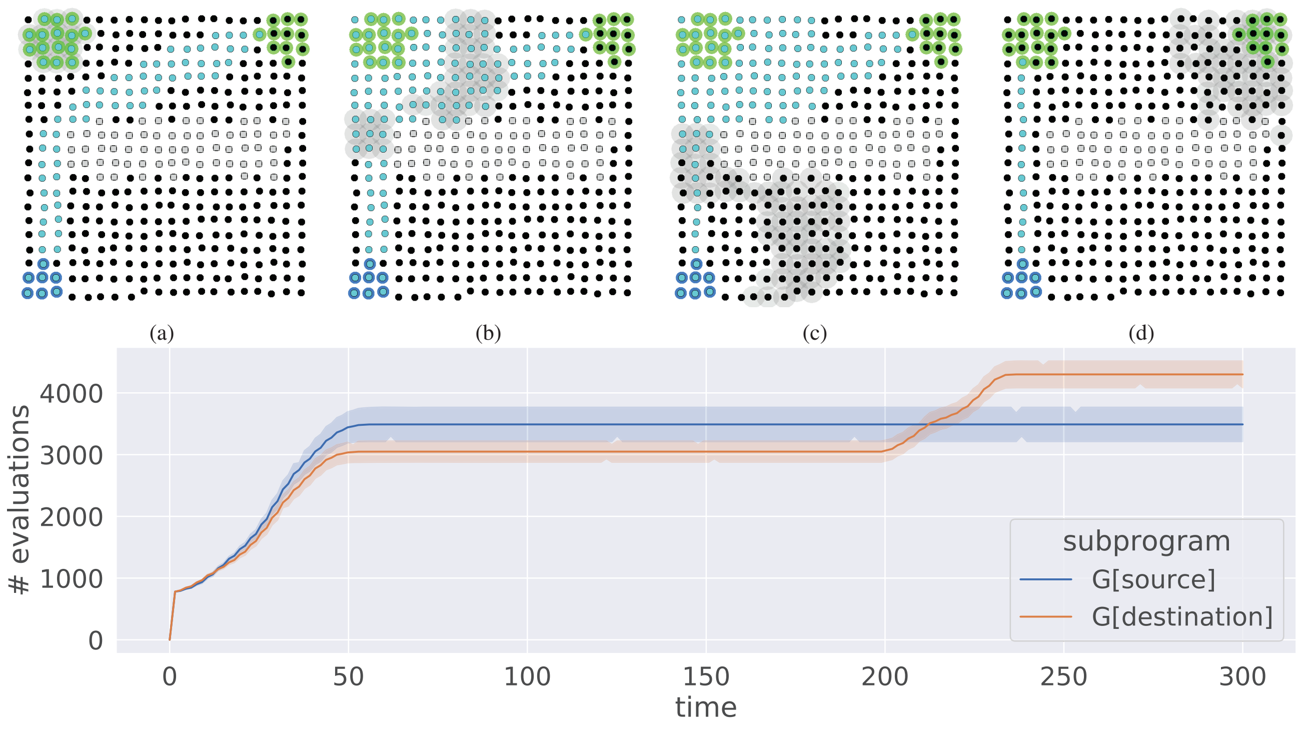


Fig. 6: The behaviour of the channel in response to changes in a destination is shown through a sequence of snapshots in the first line (using the same graphical notation as Figure 4 where, additionally, the grey shadows denote nodes where computation is taking place). It is observed that the computation "moves" to different portions of the system until it reaches a stable situation. However, the computation only results in the re-computation of the destination gradient, as seen in the bottom plot.

[22] 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

[23] K. Tumer and D. Wolpert, *Collectives and the Design of Complex Systems*. Springer, 2004. ISBN 9780387401652

[24] R. Casadei, "Artificial Collective Intelligence Engineering: A Survey of Concepts and Perspectives," *Artificial Life*, pp. 1–35, 07 2023. doi: 10.1162/artl_a_00408

[25] V. Trianni, *Evolutionary swarm robotics: evolving self-organising behaviours in groups of autonomous robots*. Springer, 2008, vol. 108.

[26] 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

[27] 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: 10.1145/1538942.1538945

[28] J. Banâtre, P. Fradet, and Y. Radenac, "Programming self-organizing systems with the higher-order chemical language," *Int. J. Unconv. Comput.*, vol. 3, no. 3, pp. 161–177, 2007.

[29] G. Aguzzi, R. Casadei, and M. Viroli, "Addressing collective computations efficiency: Towards a platform-level reinforcement learning approach," in *ACSOS*. IEEE, 2022. doi: 10.1109/ACSOS55765.2022.00019 pp. 11–20.

[30] D. Pianini, R. Casadei, M. Viroli, S. Mariani, and F. Zambonelli, "Time-fluid field-based coordination through programmable distributed schedulers," *Log. Methods Comput. Sci.*, vol. 17, no. 4, 2021. doi: 10.46298/lmcs-17(4:13)2021

[31] M. Mamei, F. Zambonelli, and L. Leonardi, "Co-fields: A physically inspired approach to motion coordination," *IEEE Pervasive Computing*, vol. 3, no. 2, pp. 52–61, 2004. doi: 10.1109/MPRV.2004.1316820

[32] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous-space programs for robotic swarms," *Neural Comput. Appl.*, vol. 19, no. 6, pp. 825–847, 2010. doi: 10.1007/s00521-010-0382-8

[33] G. Audrito, R. Casadei, F. Damiani, G. Salvaneschi, and M. Viroli, "Functional programming for distributed systems with XC," in *ECOOP*, ser. LIPIcs, vol. 222. Schloss Dagstuhl, 2022. doi: 10.4230/LIPIcs.ECOOP.2022.20 pp. 20:1–20:28.

[34] C. Pinciroli and G. Beltrame, "Buzz: An extensible programming language for heterogeneous swarm robotics," in *IROS*. IEEE, 2016. doi: 10.1109/IROS.2016.7759558 pp. 3794–3800.

[35] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, pp. 52:1–52:34, 2013. doi: 10.1145/2501654.2501666

[36] S. Blackheath, *Functional Reactive Programming*. Manning, 2016. ISBN 9781638353416

[37] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *PLDI*. ACM, 2000. doi: 10.1145/349299.349331 pp. 242–252.

[38] R. Nagpal, H. E. Shrobe, and J. Bachrach, "Organizing a global coordinate system from local information on an ad hoc sensor network," in *IPSN*, ser. LNCS, vol. 2634. Springer, 2003. doi: 10.1007/3-540-36978-3_22 pp. 333–348.

[39] G. Audrito, R. Casadei, F. Damiani, and M. Viroli, "Compositional blocks for optimal self-healing gradients," in *SASO*. IEEE, 2017. doi: 10.1109/SASO.2017.18 pp. 91–100.

[40] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[41] D. Pianini, S. Montagna, and M. Viroli, "Chemical-oriented simulation of computational systems with ALCHEMIST," *J. Simulation*, vol. 7, no. 3, pp. 202–215, 2013. doi: 10.1057/jos.2012.27

[42] C. Artho, K. Havelund, R. Kumar, and Y. Yamagata, "Domain-specific languages with Scala," in *ICFEM*, ser. LNCS, vol. 9407. Springer, 2015. doi: 10.1007/978-3-319-25423-4_1 pp. 1–16.