A Calculus of Self-stabilising Computational Fields

Mirko Viroli, Ferruccio Damiani

A Calculus of Self-Stabilising Computational Fields[*]

Mirko Viroli[1] and Ferruccio Damiani[2]

[1] University of Bologna, Italy
`mirko.viroli@unibo.it`
[2] University of Torino, Italy
`ferruccio.damiani@unito.it`

Abstract. Computational fields are spatially distributed data structures created by diffusion/aggregation processes, designed to adapt their shape to the topology of the underlying (mobile) network and to the events occurring in it: they have been proposed in a thread of recent works addressing self-organisation mechanisms for system coordination in scenarios including pervasive computing, sensor networks, and mobile robots. A key challenge for these systems is to assure behavioural correctness, namely, correspondence of micro-level specification (computational field specification) with macro-level behaviour (resulting global spatial pattern). Accordingly, in this paper we investigate the propagation process of computational fields, especially when composed one another to achieve complex spatial structures. We present a tiny, expressive, and type-sound calculus of computational fields, enjoying self-stabilisation, i.e., the ability of computational fields to react to changes in the environment finding a new stable state in finite time.

1 Introduction

Computational fields [11, 17] (sometimes simply *fields* in the following) are an abstraction traditionally used to enact self-organisation mechanisms in contexts including swarm robotics [1], sensor networks [3], pervasive computing [12], task assignment [22], and traffic control [6]. They are distributed data structures originated from pointwise events raised in some specific device (i.e., a sensor), and propagating in a whole network region until forming a spatio-temporal data structure upon which distributed and coordinated computation can take place. Example middleware/platforms supporting this notion include TOTA [12], Proto [13], and SAPERE [24, 15]. The most paradigmatic example of computational field is the so-called *gradient* [4, 12, 15], mapping each node of the network to the minimum distance from the source node where the gradient has been injected. Gradients are key to get awareness of physical/logical distances, to project a single-device event into a whole network region, and to find the direction towards certain locations of a network, e.g., for routing purposes. A number of

works have been developed that investigate coordination models supporting fields [12, 21], introduce advanced gradient-based spatial patterns [14], and develop catalogues of self-organisation mechanisms where gradients play a crucial role [8].

As with most self-organisation approaches, a key issue is to try to fill the gap between the system micro-level (the single-node computation and interaction behaviour) and the system macro-level (the shape of the globally established spatio-temporal structure), namely, ensuring that the programmed code results in the expected global-level behaviour. However, the issue of formally tackling the problem is basically yet unexplored in the context of spatial computing, coordination, and process calculi—some exceptions are [4, 9], which however apply in rather ad-hoc cases. We note instead that deepening the problem will likely shed light on which language constructs are best suited for developing well-engineered self-organisation mechanisms based on computational fields, and to consolidate existing patterns or develop new ones.

In this paper we follow this direction and address the problem of finding an expressive calculus to specify the propagation process of those computational fields for which we can identify a precise mapping between system micro- and macro-level. We identified a core calculus with sound type systems formed by three constructs only: sensor fields (considered as an environmental input), pointwise functional composition of fields, and a form of *spreading* that tightly couples information diffusion and re-aggregation. The latter is constrained so as to enforce a special "terminating progressiveness" property that we identified, by which we derive self-stabilisation [7], that is, the ability of the system running computational fields to reach a stable distributed state in spite of perturbations (changes of network topology and of local data) from which it recovers in finite time. A consequence of our results is that the ultimate (and stable) state of an even complex computational field can be fully-predicted once the environment state is known (network topology and sensors state).

The remainder of this paper is organised as follows: Section 2 presents the proposed linguistic constructs by means of examples, Section 3 provides the formal calculus, Section 4 states soundness and self-stabilisation properties, and finally Section 5 discusses related works and concludes.

2 Computational Fields

From an abstract viewpoint, a computational field is simply a map from nodes of a network to some kind of value. They are used as a valuable abstraction to engineer self-organisation into networks of situated devices. Namely, out of local interactions (devices communicating with a small neighbourhood), global and coherent patterns (the computational fields themselves) establish that are robust to changes of environmental conditions. Such an adaptive behaviour is key in developing system coordination in dynamic and unpredictable environments [16].

Self-organisation and computational fields are known to build on top of three basic mechanisms [8]: diffusion (devices broadcast information to their neighbours), aggregation (multiple information can be aggregated back into a single sum-up value), and evaporation/decay (a cleanup mechanism is used to reactively adapt to changes). For instance, these mechanisms are precisely those used to create adaptive and stable

$$e ::= x \mid v \mid s \mid g(e_1,\ldots,e_n) \mid \{e : g(@,e_1,\ldots,e_n)\} \qquad \text{expression}$$
$$g ::= f \mid o \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{function}$$
$$F ::= \text{def } T\, f(T_1\, x_1,\ldots,T_n\, x_n) \text{ is } e \qquad\qquad\qquad \text{function definition}$$

Fig. 1. Syntax of expressions and function definitions

gradients, which are building blocks of more advanced patterns [8, 14]. A gradient is used to reify in any node some information about the path towards the nearest gradient source. It can be computed by the following process: value 0 is held in the gradient source; each node executes asynchronous computation rounds in which *(i)* messages from neighbours are gathered and *aggregated* in a minimum value, *(ii)* this is increased by one and is *diffused* to all neighbours, and *(iii)* the same value is stored locally, to replace the old one which *decays*. This continuous "spreading process" stabilises to a so called *hop-count gradient*, storing distance to the nearest source in any node, and automatically repairing in finite time to changes in the environment (changes of topology, position and number of sources).

2.1 Basic Ingredients

Based on these ideas, and framing them so as to isolate those cases where the spreading process actually stabilises, we propose a tiny calculus to express computational fields. Its syntax is reported in Figure 1. Following the general approach used in other languages for spatial computing [20, 13], which the one we propose here can be considered as a core, our language is functional.

An atomic expression can be a variable x, a value v, or a sensor *s*. Variables are the formal parameters of a function. Values can be of different sorts: integers (0, 1, ... and INF meaning the maximum integer), floats (e.g., 1.0, -5.7), booleans (TRUE and FALSE), tuples (<1,TRUE>,<2,-3.5>,<1,FALSE,3>), and so on. Sensors are sources of input produced by the environment, available in each device (in examples, we shall use for them literals starting with symbol "#"). For instance, in a urban scenario we may want to use a crowd sensor #crowd yielding non-negative real numbers, to represent the perception of crowd level available in each deployed sensor over time [15].

Expressions can be composed functionally, by either a (built-in) operator o or a user-defined function f. Operators include usual mathematical/logical ones, used either in prefix or infix notation: e.g. to form expressions 2*#crowd and or(TRUE,FALSE). Operators 1st, 2nd, and so on, are used to extract the *i*-th component of a tuple. Functions are typed and can be declared by users; cyclic definitions are prohibited, and 0-ary function main is the program entry point. Types include int, float, bool, and tuple-types like <int,int>, <int,bool> and so on; each type T has a total ordered relation $\leq_T$—we use natural ordering, though in principle ad-hoc ordering relations could be used in a deployed specification language. As an example, we will use the following function restrict: def int restrict(int i, bool b) is b ? i : INF. It takes two arguments i and b, and yields the former if b is true, or INF otherwise—as we shall see, because of our semantics INF plays a role similar to an undefined value.

As in [20, 13], expressions in our language have a twofold interpretation. When focussing on the *local* device behaviour, they represent values computed in a node at a given time. When reasoning about the *global* outcome of a specification instead, they represent whole computational fields: 1 is the immutable field holding 1 in each device, #crowd is the (evolving) crowd field, and so on.

The key construct of the proposed language is *spreading*, denoted by syntax $\{e : g(@, e_1, \ldots, e_n)\}$, where e is called *source* expression, and $g(@, e_1, \ldots, e_n)$ is called *progression* expression. The latter is an expression formed by an operator/function g: if it is a function, its body should not include a spreading construct or a sensor (nor the function it calls should). Additionally, the progression expression has one hole @ playing the role of a formal argument; hence the progression expression can be seen as the body of an anonymous, unary function, which we simply call *progression*. Viewed locally to a node, expression $e = \{e_0 : g(@, e_1, \ldots, e_n)\}$ is evaluated at a given time to value v as follows:

1. expressions $e_0, e_1, \ldots, e_n$ are evaluated to values $v_0, v_1, \ldots, v_n$;
2. the current values $w_1, \ldots, w_m$ of e in neighbours are gathered;
3. for each w_j in them, the progression function is applied as $g(w_j, v_1, \ldots, v_n)$, giving value w'_j;
4. the final result v is the minimum value among $\{v_0, w'_1, \ldots, w'_m\}$: this value is made available to other nodes.

Note that $v \leq_T v_0$, and if the device is isolated then $v = v_0$. Viewed globally, $\{e_0 : g(@, e_1, \ldots, e_n)\}$ represents a field initially equal to e_0; as time passes some field values can decrease due to smaller values being received from neighbours (after applying the progressive function).

The hop-count gradient created out of a #src sensor is hence simply defined as { #src : @ + 1 }, assuming #src holds what we call a zero-field, namely, it is 0 on source nodes and INF everywhere else. In this case #src is the source expression, and g is unary successor function.

2.2 Composition Examples

As a reference scenario to ground the discussion, we can consider crowd steering in pervasive environments [15]: computational fields run on top of a myriad of small devices spread in the environment (including smartphones), and are used to guide people in complex environments (buildings, cities) towards point of interested (POIs) across appropriate paths. There, a smartphone can perceive neighbour values of a gradient spread from a POI, and give directions towards smallest values so as to steer its owner and make him/her quickly descend the gradient [12]. Starting from the hop-count gradient, various kinds of behaviour useful in crowd steering can be programmed, based on the definitions reported in Figure 2. Note that as a mere syntactic sugar, we allowed there the use of functional compositions of built-in operators and user-defined functions as progression expressions. For instance, in function gradobs, the composition of restrict and + is used. A pre-processor could easily lift out such compositions into automatically-generated functions: e.g., for gradobs it could be

```
def int grad(int i) is { i : @ + #dist }
def int restrict(int i, bool b) is b ? i : INF
def int gradobs(int i, bool b) is { i : restrict(@ + #dist, b) }
def <int,bool> sum_or(<int,bool> x, <int,bool> y) is
            <1st(x) + 1st(y), 2nd(x) or 2nd(y)>
def bool sector(int i, bool b) is  2nd({ <i, b> : sum_or(@,<#dist, b>) }
def <int,int> add_to_1st(<int,int> x, int y) is <1st(x)+ y, 2nd(x)>
def <int,int> gradcast(int i, int j) is { <i, j> : add_to_1st(@, #dist) }
def int dist(int i, int j) is gradcast(restrict(j,j==0),grad(i))
def bool path(int i, int j, int w) is grad(i)+grad(j)-w < dist(i, j)
def int channel(int i, int j, int w) is gradobs(grad(j),not path(i, j, w))
```
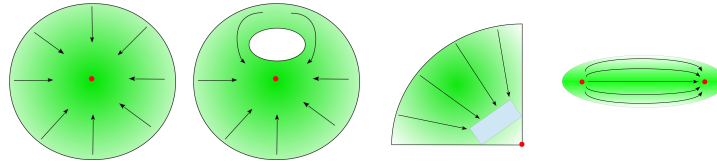
Fig. 2. Definitions of examples

Fig. 3. A pictorial representation of various fields: hop-count gradient (a), gradient circumventing "crowd" obstacles (b), sector (c), and channel (d)

"def int gradobs$lifted(int x,int y,bool b) is restrict(x + y,b)",
so the body of gradobs could become "{ i : gradobs$lifted(@,#dist,b) }"
as the syntax of our calculus actually requires.

The first function in Figure 2 defines a more powerful gradient construct, called grad, which can be used to generalise over the hop-by-hop notion of distance: sensor #dist is assumed to exist that reifies an application-specific notion of distance as a positive number. It can be 1 everywhere to model hop-count gradient, or can vary from device to device to take into consideration contextual information. For instance, it can be the output of a crowd sensor, leading to greater distances when/where crowded areas are perceived, so as to dynamically compute routes penalising crowded areas as in [15]. In this case, note that function g maps (v_1, v_2) to $v_1 + v_2$. Figure 3 (a) shows a pictorial representation, assuming devices are uniformly spread in a 2D environment: considering that an agent or data items moves in the direction descending the values of a field, a gradient looks like a sort of uniform attractor towards the source, i.e., to the nearest source node. It should be noted that when deployed in articulated environments, the gradient would stretch and dilate to accommodate the static/dynamic shape of environment, computing optimal routes.

By suitably changing the progression function, it is also possible to block the diffusion process of gradients, as shown in function gradobs: there, by restriction we turn the gradient value to INF in nodes where the "obstacle" boolean field b holds TRUE. This can be used to completely circumvent obstacle areas, as shown in Figure 3 (b). Note that

we here refer to a "blocking" behaviour, since sending a INF value has no effect on the target, and could hence be avoided for the sake of performance, e.g., not to flood the entire network. This pattern is useful whenever steering people in environments with prohibited areas—e.g. road construction in a urban scenario.

In our language it is also possible to keep track of specific situations during the propagation process, as function `sector` showcases. It takes a zero-field source `i` and a boolean field `b` denoting an area of interest: it creates a gradient of pairs, orderly holding distance from source and a boolean value representing whether the route towards the source crossed area `b`. As one such gradient is produced, it is wholly applied to operator `2nd`, extracting a sector-like boolean field as shown in Figure 3 (c). To do so, we use a special progression function `sum_or` working on `int,bool` pairs, which sums the first components, and apply disjunction to the second. This pattern is useful to make people be aware of certain areas that the proposed path would cross, so as to support proper choices among alternatives [14].

The remaining functions `gradcast`, `dist`, `path` and `channel` are used to obtain a spatial pattern more heavily relying on multi-level composition, known as *channel* [20, 13]. Assume `i` and `j` are zero-fields, and suppose to steer people in complex and large environments from area `i` to destination `j`, i.e., from a node where `i` holds 0 to a node where `j` holds 0. It is important to activate the steering service (spreading information, providing signs, and detecting contextual information such as congestion) only along the shortest path, possibly properly extended (of a distance width `w` to deal with some randomness of people movement)—see Figure 3 (d). Function `gradcast` generates a gradient, holding in each node a pair of the minimum distance to source `i` and the value of `j` in that source; this is typically used to broadcast along with a gradient a value held in its source. Function `dist` uses `gradcast` to broadcasts the distance *d* between `i` and `j`—i.e., the minimum distance between a node where `i` holds 0 and a node where `j` holds 0. This is done by sending a gradcast from the source of `j` holding the value of `grad(i)` there, which is exactly the distance *d*. Function `path` simply marks as positive those nodes whose distance from the shortest path between `i` and `j` is smaller than `w`. Finally, function `channel` generates from `j` a gradient confined inside `path(i,j,w)`, which can be used to steer people towards the POI at `j` without escaping the path area.

3   The Calculus of Self-Stabilising Computational Fields

After informally introducing the proposed calculus in previous section, we now provide a formal account of it, in order to precisely state the self-stabilisation property in next section. We first discuss typing issues in Section 3.1, then formalise the operational semantics by first focussing on single-device computations in Section 3.2, and finally on whole network evolution (Section 3.3).

3.1   Typing and Self-stabilisation

The syntax of the calculus is reported in Figure 1. As a standard syntactic notation in calculi for object-oriented and functional languages [10], we use the overbar notation to denote metavariables over lists, e.g., we let $\overline{e}$ range over lists of expressions, written

Expression typing: $\boxed{\mathscr{A} \vdash e : T}$

$$[\text{T-VAR}] \quad \frac{}{\mathscr{A}, x : T \vdash x : T} \qquad [\text{T-SNS}] \quad \frac{}{\mathscr{A} \vdash s : \textbf{typeof}(s)} \qquad [\text{T-VAL}] \quad \frac{}{\mathscr{A} \vdash v : \textbf{typeof}(v)}$$

$$[\text{T-OPFUN}] \quad \frac{signature(g) = T\,g(\overline{T}) \quad \mathscr{A} \vdash \overline{e} : \overline{T}}{\mathscr{A} \vdash g(\overline{e}) : T} \qquad [\text{T-SPR}] \quad \frac{stabilising(g) \quad \mathscr{A} \vdash g(e, \overline{e}) : T}{\mathscr{A} \vdash \{e : g(@, \overline{e})\} : T}$$

Function typing: $\boxed{\text{F OK}}$

$$[\text{T-DEF}] \quad \frac{\overline{x} : \overline{T} \vdash e : T}{\texttt{def } T\,f(\overline{T\,x}) = e \text{ OK}}$$

Fig. 4. Typing rules for expressions and function definitions

$e_1\ e_2\ \dots\ e_n$, and similarly for $\overline{x}$, $\overline{T}$ and so on. We write $[\![T]\!]$ to denote the set of the values of type T, and $signature(g)$ to denote the signature $T\,g(\overline{T})$ of g (which specifies the type T of the result and the types $\overline{T} = T_1, \dots, T_n$ of the $n \geq 0$ arguments of g).

A program **P** in our language is a mapping from function names to function definitions, enjoying the following *sanity conditions*: *(i)* $\textbf{P}(f) = \texttt{def } f \cdots (\cdots) \texttt{ is } \cdots$ for every $f \in dom(\textbf{P})$; *(ii)* for every function name f appearing anywhere in **P**, we have $f \in dom(\textbf{P})$; *(iii)* there are no cycles in the function call graph (i.e., there are no recursive functions in the program); and *(iv)* $\texttt{main} \in dom(\textbf{P})$ and it has zero arguments.

The type system we provide aims to guarantee self-stabilisation: its typing rules are given in Figure 4. *Type environments*, ranged over by $\mathscr{A}$ and written $\overline{x} : \overline{T}$, contain type assumptions for program variables. The typing judgement for expressions is of the form $\mathscr{A} \vdash e : T$, to be read: e has type T under the type assumptions $\mathscr{A}$ for the program variables occurring in e. As a standard syntax in type systems [10], given $\overline{x} = x_1, \dots, x_n$, $\overline{T} = T_1, \dots, T_n$ and $\overline{e} = e_1, \dots, e_n$ ($n \geq 0$), we write $\overline{x} : \overline{T}$ as short for $x_1 : T_1, \dots, x_n : T_n$, and $\mathscr{A} \vdash \overline{e} : \overline{T}$ as short for $\mathscr{A} \vdash e_1 : T_1 \ \cdots \ \mathscr{A} \vdash e_n : T_n$. Typing of variables, sensors, values, built-in operators and user-defined functions application are almost standard (in particular, values and sensors are given a type by construction). The only ad-hoc typing is provided for spreading expressions $\{e : g(@, \overline{e})\}$: they are trivially given the same type of $g(e, \overline{e})$, though additional conditions has to be checked to guarantee self-stabilisation, which are at the core of the technical result provided in this paper. In particular, any function g used in a spreading expression must be a *stabilising progression function*, according to the following definition.

Definition 1 (Stabilising progression). A function g with signature $T\,g(T_1, \dots, T_m)$ is a *stabilising progression* (notation $stabilising(g)$) if the following conditions hold:

(i) $m > 0$ and $T = T_1$;
(ii) g is a so-called *pure operator*, namely, it is either a built-in operator o, or a user-defined function f whose call graph (including f itself) does not contain functions with spreading expressions or sensors in their body: in this case, we write $[\![g]\!]$ to denote the trivial mapping that provides the semantics of g symbol to a function;
(iii) T is so-called *locally noetherian*, to mean that $[\![T]\!]$ is equipped with a total order relation $\leq_T$, and for every element $v \in [\![T]\!]$, there are no infinite ascending chains of elements $v_0 <_T v_1 <_T v_2 \cdots$ such that (for every $n \geq 0$) $v_n <_T v$;

(iv) g is *monotone* in its first argument, i.e., $v \leq_T v'$ implies $[\![g]\!](v, \overline{v}) \leq_T [\![g]\!](v', \overline{v}))$ for any $\overline{v}$;

(v) g is *progressive* in its first argument, i.e.,
- if $[\![T]\!]$ has not a maximum element,[3] it holds that: $v <_T [\![g]\!](v, \overline{v})$ for any $\overline{v}$;
- if $[\![T]\!]$ has a maximum element[4] written $\mathtt{top(T)}$, it holds that $[\![g]\!](\mathtt{top(T)}, \overline{v}) = \mathtt{top(T)}$ and, for all $v \in [\![T]\!] - \{\mathtt{top(T)}\}$, $v <_T [\![g]\!](v, \overline{v})$.

Function typing (represented by judgement "F OK") is standard. Then, in the following we always consider a *well-typed* program **P**, to mean that all the function declarations in **P** are well typed.

Note that all examples provided in previous section amount to well typed functions, with few inessential caveats. First, as already discussed, in spreading expressions we use compositions of functions: this is legitimate since it is easy to see that composition of stabilising progressions is stabilising. Second, more refined types are needed to correctly identify certain spreading expressions as stabilising. For instance, in function `grad`, the sensor `#dist` must have a positive integer type (e.g., `posint`), and operator `+` should be replaced by a sum operator that accepts a positive number only on right (e.g., `+<int,posint>`), and similarly for other cases. Third, to correctly type-check the functions that use tuples (which have not been explicitly modelled in the calculus) one would need to consider a polymorphic type system a la ML in the usual way. Handling all these advanced typing aspects, as well as presenting the formalisation of the *stabilising*($\cdot$) predicate (that is, an algorithm to check whether the conditions for a function to be stabilising hold), has not been considered here for the sake of space and since they are orthogonal aspects.

3.2 Device Computation

In the following, we let meta-variables ι and κ range over the denumerable set **I** of *device identifiers*, meta-variable I over finite sets of such devices, meta-variables u, v and w over values. Given a finite nonempty set $V \subseteq [\![T]\!]$ we denote by $\bigwedge V$ its minimum element, and write $v \wedge v'$ as short for $\bigwedge \{v, v'\}$.

To simplify the notation, we shall assume a fixed program **P** and write $\mathtt{e_{main}}$ to denote the body of the `main` function. We say that "device ι *fires*", to mean that expression $\mathtt{e_{main}}$ is evaluated on device ι. The result of evaluation is a *value-tree*, which is an ordered tree of values, tracking the result of any evaluated subexpression. Intuitively, such evaluation is performed against the value-trees of neighbours and the current value of sensors, and produces as result a new value-tree that is conversely made available to other neighbours for their firing.[5] The syntax of value-trees is given in Figure 5, together with the definition of the auxiliary functions $\rho(\cdot)$ and $\pi_i(\cdot)$ for extracting the root value and the *i*-th subtree of a value-tree, respectively—also the extension of these

---

[3] Like, e.g., the `BigInteger` type in JAVA.

[4] Like, e.g., the the `double` type in JAVA, which has top element `Double.POSITIVE_INFINITY`.

[5] Accordingly, since a function g used in a spreading expression $\{\mathtt{e_0} : \mathtt{g(@, e_1, \ldots, e_n)}\}$ must be a pure operator (cf. Section 3.1), only the root of the produced sub-tree must be stored (c.f. rule [E-SPR]). Also, note that any implementation might massively compress the value-tree, storing only enough information for spreading expressions to be aligned.

functions to sequences of value-environments $\overline{\theta}$ is defined. We sometimes abuse the notation writing a value-tree with just the root as v instead of $v()$. The state of sensors σ is a map from sensor names to values, modelling the inputs received from the external world. This is written $\overline{s} \triangleright \overline{v}$ as an abuse of notation to mean $s_1 \triangleright v_1, \ldots s_n \triangleright v_n$. We shall assume that it is complete (it has a mapping for any sensor used in the program), and correct (each sensor s has a type written **typeof**(s), and is mapped to a value of that type). For this map, and for the others to come, we shall use the following notations: $\sigma(s)$ is used to extract the value that s is mapped to, $\sigma[\sigma']$ is the map obtained by updating σ with all the associations $s \triangleright v$ of σ' which do not escape the domain of σ (namely, only those such that σ is defined for s).

The computation that takes place on a single device is formalised by the big-step operational semantics rules given in Figure 5. The derived judgements are of the form $\sigma; \overline{\theta} \vdash e \Downarrow \theta$, to be read "expression e evaluates to value-tree θ on sensor state σ and w.r.t. the value-trees $\overline{\theta}$", where:

- σ is the current sensor-value map, modelling the inputs received from the external world;
- $\overline{\theta}$ is the list of the value-trees produced by the most recent evaluation of e on the current device's neighbours;
- e is the closed expression to be evaluated;
- the value-tree θ represents the values computed for all the expressions encountered during the evaluation of e— in particular $\rho(\theta)$ is the local value of field expression e.

The rules of the operational semantics are *syntax directed*, namely, the rule used for deriving a judgement $\sigma; \overline{\theta} \vdash e \Downarrow \theta$ is univocally determined by e (cf. Figure 5). Therefore, the shape of the value-tree θ is univocally determined by e, and the whole value-tree is univocally determined by σ, $\overline{\theta}$, and e.

The rules of the operational semantics are almost standard, with the exception that rules [E-OP], [E-FUN] and [E-SPR] use the auxiliary function $\pi_i(\cdot)$ to ensure that, in the judgements in the premise of the rule, the value-tree environment is aligned with the expression to be evaluated.

The most important rule is [E-SPR] which handles spreading expressions formalising the description provided in Section 2.1. It first recursively evaluates expressions e_i to value-trees η_i (after proper alignment of value-tree environment by operator $\pi_i(.)$) and top-level values v_i. Then it gets from neighbours their values w_j for the spreading expression, and for each of them g is evaluated giving top-level result w_j. The resulting value is then obtained by the minimum among v_0 and the values w_j (which equates to v_0 if there are currently no neighbours).

3.3 Network Evolution

We now provide an operational semantics for the evolution of whole networks, namely, for modelling the distributed evolution of computational fields over time. Figure 6 (top) defines key syntactic elements to this end. F models the overall computational field (state), as a map from device identifiers to value-trees. τ models *network topology*,

Value-trees and sensor-value maps:

$\theta, \eta ::= \mathtt{v}(\overline{\theta})$ value-tree

$\sigma ::= \overline{s} \triangleright \overline{\mathtt{v}}$ sensor-value map

Auxiliay functions:

$\rho(\mathtt{v}(\overline{\theta})) = \mathtt{v}$ $\pi_i(\mathtt{v}(\theta_1, \ldots, \theta_n)) = \theta_i$

$\rho(\theta_1, \ldots, \theta_n) = \rho(\theta_1), \ldots, \rho(\theta_n)$ $\pi_i(\theta_1, \ldots, \theta_n) = \pi_i(\theta_1), \ldots, \pi_i(\theta_n)$

Rules for expression evaluation: $\boxed{\sigma; \overline{\theta} \vdash \mathtt{e} \Downarrow \theta}$

[E-SNS]
$$\frac{}{\sigma; \overline{\theta} \vdash s \Downarrow \sigma(s)}$$

[E-VAL]
$$\frac{}{\sigma; \overline{\theta} \vdash \mathtt{v} \Downarrow \mathtt{v}}$$

[E-OP]
$$\frac{\iota; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \eta_1 \quad \cdots \quad \sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \eta_n \quad \mathtt{v} = [\![\mathtt{o}]\!](\rho(\eta_1), \ldots, \rho(\eta_n))}{\sigma; \overline{\theta} \vdash \mathtt{o}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \Downarrow \mathtt{v}(\eta_1, \ldots, \eta_n)}$$

[E-FUN]
$$\frac{\mathtt{def}\ \mathtt{T}\ \mathtt{f}(\mathtt{T}_1\ \mathtt{x}_1, \ldots, \mathtt{T}_n\ \mathtt{x}_n) = \mathtt{e} \quad \sigma; \pi_1(\overline{\theta}) \vdash \mathtt{e}_1 \Downarrow \eta_1 \quad \cdots \quad \sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \eta_n \quad \sigma; \pi_{n+1}(\overline{\theta}) \vdash \mathtt{e}[\mathtt{x}_1 := \rho(\theta_1') \quad \ldots \quad \mathtt{x}_n := \rho(\theta_n')] \Downarrow \mathtt{v}(\overline{\eta})}{\sigma; \overline{\theta} \vdash \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \Downarrow \mathtt{v}(\theta_1', \ldots, \theta_n', \mathtt{v}(\overline{\eta}))}$$

[E-SPR]
$$\frac{\sigma; \pi_0(\overline{\theta}) \vdash \mathtt{e}_0 \Downarrow \eta_0 \quad \cdots \quad \sigma; \pi_n(\overline{\theta}) \vdash \mathtt{e}_n \Downarrow \eta_n \quad \rho(\eta_0, \ldots, \eta_n) = \mathtt{v}_0 \ldots \mathtt{v}_n \quad \rho(\overline{\theta}) = \mathtt{w}_1 \ldots \mathtt{w}_m \quad \sigma; \emptyset \vdash \mathtt{g}(\mathtt{w}_1, \mathtt{v}_1, \ldots, \mathtt{v}_n) \Downarrow \mathtt{u}_1(\cdots) \quad \ldots \quad \sigma; \emptyset \vdash \mathtt{g}(\mathtt{w}_m, \mathtt{v}_1, \ldots, \mathtt{v}_n) \Downarrow \mathtt{u}_m(\cdots)}{\sigma; \overline{\theta} \vdash \{\mathtt{e}_0 : \mathtt{g}(@, \mathtt{e}_1, \ldots, \mathtt{e}_n)\} \Downarrow \bigwedge\{\mathtt{v}_0, \mathtt{u}_1, \ldots, \mathtt{u}_m\}(\eta_0, \eta_1, \ldots, \eta_n)}$$

Fig. 5. Big-step operational semantics for expression evaluation

namely, a directed neighbouring graph, as a map from device identifiers to set of identifiers. Σ models *sensor (distributed) state*, as a map from device identifiers to (local) sensors (i.e., sensor name/value maps). Then, E (a couple of topology and sensor state) models the system's environment. So, a whole network configuration N is a couple of a field and environment.

We define network operational semantics in terms of small-steps transitions of the kind $N \xrightarrow{\ell} N'$, where ℓ is either a device identifier in case it represents its firing, or label ε to model any environment change. This is formalised in Figure 6 (bottom). Rule [N-FIR] models a computation round (firing) at device ι: it reconstructs the proper local environment, taking local sensors ($\Sigma(\iota)$) and accessing the value-trees of ι's neighbours; then by the single device semantics we obtain the device's value-tree θ, which is used to update system configuration. Rule [N-ENV] takes into account the change of the environment to a new well-formed environment E'. Let $\iota_1, \ldots, \iota_n$ be the domain of E'. We first construct a field F_0 associating to all the devices of E' the default value-trees $\theta_1, \ldots, \theta_n$ obtained by making devices perform an evaluation with no neighbours and sensors as of E'. Then, we adapt the existing field F to the new set of devices: $F_0[F]$ automatically handles removal of devices, map of new devices to their default value-tree, and retention of existing value-trees in the other devices.

Upon this semantics. we introduce the following definitions and notations:

Initiality The empty network configuration $\langle \emptyset \triangleright \emptyset, \emptyset \triangleright \emptyset; \emptyset \triangleright \emptyset \rangle$ is said *initial*.

System configurations and action labels:

$$F ::= \bar{\iota} \triangleright \bar{\theta} \qquad\qquad\qquad \text{computational field}$$
$$\tau ::= \bar{\iota} \triangleright \bar{I} \qquad\qquad\qquad\qquad \text{topology}$$
$$\Sigma ::= \bar{\iota} \triangleright \bar{\sigma} \qquad\qquad\qquad\qquad \text{sensors-map}$$
$$E ::= \tau, \Sigma \qquad\qquad\qquad\qquad \text{environment}$$
$$N ::= \langle E; F \rangle \qquad\qquad\qquad \text{network configuration}$$
$$\ell ::= \iota \mid \varepsilon \qquad\qquad\qquad\qquad \text{action label}$$

Environment well-formedness:

$WFE(\tau, \Sigma)$ holds if τ, Σ have same domain, and τ's values do not escape it.

Transition rules for network evolution: $\qquad\qquad\boxed{N \xrightarrow{\ell} N}$

$$\text{[N-FIR]} \frac{E = \tau, \Sigma \qquad \tau(\iota) = \bar{\iota} \qquad \Sigma(\iota); F(\bar{\iota}) \vdash \mathtt{e_{main}} \Downarrow \theta}{\langle E; F \rangle \xrightarrow{\iota} \langle E; F[\iota \triangleright \theta] \rangle}$$

$$\text{[N-ENV]}$$
$$\frac{\begin{array}{c} WFE(E') \qquad E' = \tau, \iota_1 \triangleright \sigma_1, \ldots, \iota_n \triangleright \sigma_n \\ \sigma_1; \emptyset \vdash \mathtt{e_{main}} \Downarrow \theta_1 \quad \cdots \quad \sigma_n; \emptyset \vdash \mathtt{e_{main}} \Downarrow \theta_n \qquad F_0 = \iota_1 \triangleright \theta_1, \ldots, \iota_n \triangleright \theta_n \end{array}}{\langle E; F \rangle \xrightarrow{\varepsilon} \langle E'; F_0[F] \rangle}$$

Fig. 6. Small-step operational semantics for network evolution

Reachability Write $N \xRightarrow{\bar{\ell}} N'$ as short for $N \xrightarrow{\ell_1} N_1 \xrightarrow{\ell_2} \cdots \xrightarrow{\ell_n} N'$: a configuration N is said *reachable* if $N_0 \xRightarrow{\bar{\ell}} N$ where N_0 is initial. Reachable configurations are the well-formed ones, and in the following we shall implicitly consider only reachable configurations.

Firing A firing evolution from N to N', written $N \Longrightarrow N'$, is one such that $N \xRightarrow{\bar{\iota}} N'$ for some $\bar{\iota}$, namely, where only firings occur.

Stability A system state N is said *stable* if $N \xrightarrow{\iota} N'$ implies $N = N'$, namely, the computation of fields reached a fixpoint in the current environment. Note that if N is stable, then it also holds that $N \Longrightarrow N'$ implies $N = N'$.

Fairness We say that a sequence of device fires is *k-fair* ($k \geq 0$) to mean that, for every h ($1 \leq h \leq k$), the h-th fire of any device is followed by at least $k - h$ fires of all the other devices. Accordingly, a firing evolution $N \xRightarrow{\bar{\iota}} N'$ is said *k-fair*, written $N \xRightarrow{\bar{\iota}}_k N'$, to mean that $\bar{\iota}$ is k-fair. We also write $N \Longrightarrow_k N'$ if $N \xRightarrow{\bar{\iota}}_k N'$ for some $\bar{\iota}$. This notion of fairness will be used to characterise finite firing evolutions in which all devices are given equal chance to fire when all others had.

Self-stabilisation A system state $\langle E; F \rangle$ is said to *self-stabilise* to $\langle E; F' \rangle$ if there is a $k > 0$ and a field F' such that $\langle E; F \rangle \Longrightarrow_k \langle E; F' \rangle$ implies $\langle E; F' \rangle$ is stable, and F' is univocally determined by E. Self-stability basically amounts to the inevitable reachability of a stable state depending only on environment conditions, through a sufficiently long fair evolution. Hence, the terminology is abused equivalently saying that a field expression $\mathtt{e_{main}}$ is self-stabilising if for any environment state E there exists a unique stable field F' such that *any* $\langle E; F \rangle$ *self-stabilises* to $\langle E; F' \rangle$.

3.4 An example application of the semantics

Consider the function definition `def int main() is { #src : @ + #dist }`, where `#src` is a sensor of type `int` (with default value 0), `#dist` is a sensor of type `posint` (positive integers, with default value 1) and `+` is a built-in sum operator which can be given signature `int +(int,posint)`. Note that operator `+` (which is the progression function used in this spreading expression) is a self-stabilising progression, according to the definition in 3.1.

Starting from an initial empty configuration, we move by rule [N-ENV] to a new environment with the following features:

- the domain is formed by $2n$ ($n \geq 1$) devices $\iota_1, \ldots, \iota_n, \iota_{n+1}, \ldots, \iota_{2n}$;
- the topology is such that any device ι_i is connected to ι_{i+1} and ι_{i-1} (if they exist);
- sensor `#dist` gives 1 everywhere;
- sensor `#src` gives 0 on the devices ι_i ($1 \leq i \leq n$, briefly referred to as *left devices*) and a value u ($u > n+1$) on the devices ι_j ($n+1 \leq j \leq 2n$, briefly referred to as *right devices*).

Accordingly, the left devices are all assigned to value-tree $0(0,1)$, while the right ones to $u(u,1)$: hence, the resulting field maps left devices to 0 and right devices to 1—remember such evaluations are done assuming nodes are isolated, hence the result is exactly the value of the source expression. With this environment, the firing of a device can only replace the root of a value-tree, making it the minimum of the source expression's value and the minimum of the successor of neighbour's values. Hence, any firing of a device that is not ι_{n+1} does not change its value-tree. When ι_{n+1} fires instead by rule [N-FIR], its value-tree becomes $1(u,1)$, and it remains so if more firings occur next.

Now, only a firing at ι_{n+2} causes a change: its value-tree becomes $2(u,1)$. Going on this way, it easy to see that after any n-fair firing sequence the network self-stabilises to the field state where left devices still have value-tree $0(u,1)$, while right devices $\iota_{n+1}, \iota_{n+2}, \iota_{n+3}, \ldots$ have value-trees $1(u,1), 2(u,1), 3(u,1), \ldots$, respectively. That is, the root of such trees form a hop-count gradient, measuring minimum distance to the source nodes, namely, the left devices.

It can also be shown that any environment change, followed by a sufficiently long firing sequence, makes the system self-stabilise again, possibly to a different field state. For instance, if the two connections of ι_{2n-1} to/from ι_{2n-2} break (assuming $n > 2$), the part of the network excluding ι_{2n-1} and ι_{2n} keeps stable in the same state. The values at ι_{2n-1} and ι_{2n} start raising instead, increasing of 2 alternatively until both reach the initial value-trees $u(u,1)$—and this happens in finite time by a fair evolution thanks to the local noetherianity property of stabilising progressions. Note that the final state is still the hop-count gradient, though adapted to the new environment topology.

An example of field that is *not* self-stabilising is `{ #src : @ }`: there, progression function is the identity, which is not a stabilising progression (cf. Definition 1). Assuming a connected network, and `#src` holding value v_s in one node and `top(int)` in all others, then *any* configuration where all nodes hold the same value v less than or equal to v_s is trivially stable. This would model a source gossiping a fixed value v_s everywhere: if the source suddenly gossips a value v_s' smaller than v, then the network would self-organise and all nodes would eventually hold v_s'. However, if the source then

gossips a value v''_s greater than v'_s, the network would *not* self-organise and all nodes would remain stuck to value v'_s.

4 Properties

In this section we state the main property of the proposed calculus, namely, self-stabilisation. Few preliminaries and results are given first. Given an expression e such that $\overline{x} : \overline{T} \vdash e : T$, the set $WFVT(\overline{x} : \overline{T}, e, T)$ of the *well-formed* value-trees for e, is inductively defined as follows: $\theta \in WFVT(\overline{x} : \overline{T}, e, T)$ if there exist

- a sensor mapping σ,
- well-formed tree environments $\overline{\theta} \in WFVT(\overline{x} : \overline{T}, e, T)$; and
- values $\overline{v}$ such that $length(\overline{v}) = length(\overline{x})$ and $\emptyset \vdash \overline{v} : \overline{T}$;

such that $\sigma; \overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$ holds. As this notion is defined we can state the following two theorems, guaranteeing that from a properly typed environment, evaluation of a well-typed expression yields a properly typed result and always terminates, respectively.

Theorem 1 (Device computation type preservation). If $\overline{x} : \overline{T} \vdash e : T$, σ is a sensor mapping, $\overline{\theta} \in WFVT(\overline{x} : \overline{T}, e, T)$, $length(\overline{v}) = length(\overline{x})$, $\emptyset \vdash \overline{v} : \overline{T}$ and $\sigma; \overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$, then $\emptyset \vdash \rho(\theta) : T$.

Proof (sketch). By induction on the application of the rules in Fig. 5 (by observing that, in rules [E-OP], [E-FUN] and [E-GRD], the use of the auxiliary function $\pi_i(\cdot)$ preserves the well formedness of the value-trees $\overline{\theta}$).

Theorem 2 (Device computation termination). If $\overline{x} : \overline{T} \vdash e : T$, σ is a sensor mapping, $\overline{\theta} \in WFVT(\overline{x} : \overline{T}, e, T)$, $length(\overline{v}) = length(\overline{x})$ and $\emptyset \vdash \overline{v} : \overline{T}$, then $\sigma; \overline{\theta} \vdash e[\overline{x} := \overline{v}] \Downarrow \theta$ for some value-tree θ.

Proof (sketch). By induction on the syntax of expressions and on the number of function calls that may be encountered during the evaluation of the closed expression $e[\overline{x} := \overline{v}]$ (cf. sanity condition *(iii)* in Section 3.1).

The two theorems above basically state soundness and termination of local computations, that is, from a well-typed input computation completes without errors. On top of them we state the main technical result of the paper, namely, self-stabilisation of any well-constructed field expression in any environment.

Theorem 3 (Network self-stabilisation). Given a well-typed program, any reachable network configuration $\langle F; E \rangle$ self-stabilises.

Proof (sketch). By induction on the syntax of closed expressions e and on the number of function calls that may be encountered during the evaluation of e. Let F_e denote the computation field associated to the closed expression e, so $F = F_{e_{main}}$. The idea is to prove the following auxiliary statements:

1. For every network configuration N, there exists $k \geq 0$ such that: $N \Longrightarrow_k N'$ implies N' is stable.

2. For every network configuration $\langle F_e; E \rangle$, there exist a stable field F_e' and an evolution $\langle F_e; E \rangle \Longrightarrow_h \langle F_e'; E \rangle$ $(h \geq 0)$ such that: *(i)* F_e' is univocally determined by E; and *(ii)* for every stable network configuration $\langle F_e''; E \rangle$ it holds that $F_e'' = F_e'$.

For both the statements, the key case of the proof is that of a spreading expression, $e = \{e_0 : g(@, e_1, \ldots, e_n)\}$, which exploits the following auxiliary results: *(i)* If e_0 stabilises to F_{e_0}, and $\langle F_e; E \rangle \Longrightarrow_1 \langle F_e'''; E \rangle$ then the field F_e''' is *pre-stable*, i.e., for every device ι it holds that $F_e'''(\iota) \leq F_{e_0}'''(\iota) = F_{e_0}(\iota)$; *(ii)* Pre-stability is preserved by firing evolution (i.e., if N_1 is pre-stable and $N_1 \Longrightarrow N_2$, then N_2 is pre-stable); and *(iii)* Every stable network configuration is pre-stable. Moreover, statement 2 above is proved by: *(i)* Building an evolution $\langle F_e; E \rangle \Longrightarrow_1 \langle F_e'''; E \rangle \Longrightarrow_{h-1} \langle F_e'; E \rangle$ together with a set of stable devices $\mathbf{S}$ such that: *(i.a)* at the beginning of the evolution the set $\mathbf{S}$ is empty; *(i.b)* at the end of the evolution the set $\mathbf{S}$ contains all the devices of the network; *(i.c)* during the construction of the evolution, if a device ι is added to $\mathbf{S}$, then ι is stable, its value is the minimum about the values of the devices $\notin \mathbf{S}$ both in the current network configuration and in the final network configuration $\langle F_e'; E \rangle$, and that value is univocally determined by E; and *(ii)* Showing that, for any stable network configuration $\langle F_e''; E \rangle$, if the devices fire in the same order they fire in the evolution $\langle F_e'''; E \rangle \Longrightarrow_{h-1} \langle F_e'; E \rangle$ than each device must assume the same value it has in F_e'. So, since $\langle F_e''; E \rangle$ is stable, it must hold that $F_e'' = F_e'$. The construction of the evolution $\langle F_e'''; E \rangle \Longrightarrow_{h-1} \langle F_e'; E \rangle$ exploits the fact that g is a stabilising progression and that $\langle F_e'''; E \rangle$ is pre-stable.

The fact that any well-typed program self-stabilises in any well-formed environment independently of any intermediate computation state is a result of key importance. It means that any well-typed expression can be associated to a final and stable field, reached in finite time by fair evolutions and adapting to the shape of the environment. This acts as the sought bridge between the micro-level (field expression in program code), and the macro-level (expected global outcome).

5 Conclusion, Related and Future Work

This paper aims at contributing to the general problem of identifying sound techniques for engineering self-organising applications. In particular: we introduce a tiny yet expressive calculus of computational fields, we show how it can model several spatial patterns of general interest (though focussing on examples of crowd steering scenarios in ad-hoc networks) and then prove self-stabilisation. Some of the material presented here was informally sketched in [18]: the present paper fully develops the idea, providing a type-sound calculus, a precise definitions of self-stabilisation, and proved sufficient conditions for self-stabilisation.

The problem of identifying self-stabilising algorithms in distributed systems is a long investigated one [7]. Creating a hop-count gradient is considered as a preliminary step in the creation of the spanning tree of a graph in [7]: an algorithm known to self-stabilise. Our main novelty in this context is that self-stabilisation is not proved for a specific algorithm/system: it is proved for all fields inductively obtained by functional composition of fixed fields (sensors, values) and by a gradient-inspired spreading process. As argued in [8], there is a whole catalogue of self-organisation patterns can be derived this way.

To the best of our knowledge, the only work aiming at a mathematical proof of stabilisation for the specific case of computational fields is [4]. There, a self-healing gradient algorithm called CRF (constraints and restoring forces) is introduced to estimate physical distance in a spatial computer, where the neighbouring relation is fixed to unit-disc radio, and node firing is strictly connected to physical time. Compared to our approach, the work in [4] tackles a more specific problem, and is highly dependent on the underlying spatial computer assumptions.

Our work is aimed to find applications to a number of models, languages, and architectures rooted on spatial computations and computational fields, a thorough review of which may be found in [5]. Examples of such models include the Hood sensor network abstraction [23], the $\sigma\tau$-Linda model [21], the SAPERE computing model [15], and TOTA middleware [12], which all implement computational fields using similar notions of spreading. More generally, Proto [13, 3] and its formalisation [19, 20], provides a functional model which served as a starting point for our approach. Proto is based on a wider set of constructs than the one we proposed, though, which makes it very hard to formally address general self-stabilisation properties. In particular, it was key to our end to neglect recursive function calls (in order to ensure termination of device fires), stateful operations (in our model, the state of a device is always cleaned up before computing the new one), and to restrict aggregation to minimum function and progression to what we called "self-stabilising" functions. In its current form, we believe our result already implies self-stabilisation of certain Proto fields, like those intertwining constructs `rep` (state), `nbr` (access to neighbours), and `min-hood+` (min-aggregation) as follows: `(rep x (inf) (min F (g (min-hood+ (nbr x)) F`$_1$` .. F`$_n$`)))`. One such connection, however, needs to be formally addressed in future works, along with the possibility of widening the applicability of our result by releasing some assumption. Additionally, we plan to develop an algorithm to check whether the progression function at hand is actually self-stabilising. Another interesting future thread concerns finding a characterisation of expressiveness of computational field mechanisms and spatial computing languages [2], with clear implications in the design of new mechanisms.

References

1. J. Bachrach, J. Beal, and J. McLurkin. Composable continuous space programs for robotic swarms. *Neural Computing and Applications*, 19(6):825–847, 2010.
2. J. Beal. A basis set of operators for space-time computations. In *Spatial Computing Workshop*, 2010. Available at: `http://www.spatial-computing.org/scw10/`.
3. J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.
4. J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975. ACM, 2008.
5. J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of*

Domain-Specific Languages: Recent Developments, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`.

6. R. Claes, T. Holvoet, and D. Weyns. A decentralized approach for anticipatory vehicle routing using delegate multiagent systems. *IEEE Transactions on Intelligent Transportation Systems*, 12(2):364–373, 2011.

7. S. Dolev. *Self-Stabilization*. MIT Press, 2000.

8. J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.

9. J.-L. Giavitto, O. Michel, and A. Spicher. Spatial organization of the chemical paradigm and the specification of autonomic systems. volume 5380 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2008.

10. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

11. B. MacLennan. Field computation: A theoretical framework for massively parallel analog computation, parts i-iv. Technical Report Department of Computer Science Technical Report CS-90-100, University of Tennessee, Knoxville, February 1990.

12. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.

13. MIT Proto. software available at `http://proto.bbn.com/`, Retrieved January 1st, 2012.

14. S. Montagna, D. Pianini, and M. Viroli. Gradient-based self-organisation patterns of anticipative adaptation. In *Proceedings of SASO 2012*, pages 169–174. IEEE, September 2012.

15. S. Montagna, M. Viroli, J. L. Fernandez-Marquez, G. Di Marzo Serugendo, and F. Zambonelli. Injecting self-organisation into pervasive service ecosystems. *Mobile Networks and Applications*, pages 1–15, September 2012. Online first.

16. A. Omicini and M. Viroli. Coordination models and languages: From parallel computing to self-organisation. *The Knowledge Engineering Review*, 26(1):53–59, Mar. 2011.

17. M. Tokoro. Computational field model: toward a new computing model/methodology for open distributed environment. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*, pages 501–506, 1990.

18. M. Viroli. Engineering confluent computational fields: from functions to rewrite rules. In *Spatial Computing Workshop (SCW 2013)*, AAMAS 2013, May 2013.

19. M. Viroli, J. Beal, and K. Usbeck. Operational semantics of Proto. *Science of Computer Programming*, 78(6):633–656, June 2013.

20. M. Viroli, F. Damiani, and J. Beal. A calculus of computational fields. In C. Canal and M. Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.

21. M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Proceedings of Coordination 2012*, volume 7274 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2012.

22. D. Weyns, N. Boucké, and T. Holvoet. A field-based versus a protocol-based approach for adaptive task assignment. *Autonomous Agents and Multi-Agent Systems*, 17(2):288–319, 2008.

23. K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.

24. F. Zambonelli, G. Castelli, L. Ferrari, M. Mamei, A. Rosi, G. D. M. Serugendo, M. Risoldi, A.-E. Tchao, S. Dobson, G. Stevenson, J. Ye, E. Nardini, A. Omicini, S. Montagna, M. Viroli, A. Ferscha, S. Maschek, and B. Wally. Self-aware pervasive service ecosystems. *Procedia CS*, 7:197–199, 2011.