Engineering resilient collective adaptive systems by self-stabilisation

Engineering Resilient Collective Adaptive Systems by Self-Stabilisation

MIRKO VIROLI, Università di Bologna, Italy
GIORGIO AUDRITO, Università di Torino, Italy
JACOB BEAL, Raytheon BBN Technologies, USA
FERRUCCIO DAMIANI, Università di Torino, Italy
DANILO PIANINI, Università di Bologna, Italy

Collective adaptive systems are an emerging class of networked computational systems, particularly suited for application domains such as smart cities, complex sensor networks, and the Internet of Things. These systems tend to feature large scale, heterogeneity of communication model (including opportunistic peer-to-peer wireless interaction), and require inherent self-adaptiveness properties to address unforeseen changes in operating conditions. In this context, it is extremely difficult (if not seemingly intractable) to engineer reusable pieces of distributed behaviour so as to make them provably correct and smoothly composable.

Building on the field calculus, a computational model (and associated toolchain) capturing the notion of aggregate network-level computation, we address this problem with an engineering methodology coupling formal theory and computer simulation. On the one hand, functional properties are addressed by identifying the largest-to-date field calculus fragment generating self-stabilising behaviour, guaranteed to eventually attain a correct and stable final state despite any transient perturbation in state or topology, and including highly reusable building blocks for information spreading, aggregation, and time evolution. On the other hand, dynamical properties are addressed by simulation, empirically evaluating the different performances that can be obtained by switching between implementations of building blocks with provably equivalent functional properties. Overall, our methodology sheds light on how to identify core building blocks of collective behaviour, and how to select implementations that improve system performance while leaving overall system function and resiliency properties unchanged.

CCS Concepts: • **Theory of computation** → **Self-organization**; • **Software and its engineering** → **Specification languages**; *Semantics*; *Software design engineering*; *Empirical software validation*; • **Computing methodologies** → *Model verification and validation*;

Additional Key Words and Phrases: Aggregate computing; Field calculus; Collective adaptive systems; Self-stabilisation; Distributed algorithms, Simulation and Modeling

ACM Reference Format:
MIRKO VIROLI, GIORGIO AUDRITO, JACOB BEAL, FERRUCCIO DAMIANI, and DANILO PIANINI. 2018. Engineering Resilient Collective Adaptive Systems by Self-Stabilisation. *ACM Trans. Model. Comput. Simul.* 0, 0, Article 0 (2018), 28 pages. https://doi.org/0000001.0000001

Authors' addresses: MIRKO VIROLI, Università di Bologna, Cesena, Italy, mirko.viroli@unibo.it; GIORGIO AUDRITO, Università di Torino, Torino, Italy, giorgio.audrito@di.unito.it; JACOB BEAL, Raytheon BBN Technologies, Cambridge (MA), USA, jakebeal@ieee.org; FERRUCCIO DAMIANI, Università di Torino, Torino, Italy, ferruccio.damiani@di.unito.it; DANILO PIANINI, Università di Bologna, Cesena, Italy, danilo.pianini@unibo.it.

XXXX-XXXX/2018/0-ART0 $15.00
https://doi.org/0000001.0000001

1 INTRODUCTION

Collective adaptive systems are an emerging class of networked computational systems situated in the real-world, finding extensive application in domains such as smart cities, complex sensor networks, and the Internet of Things. The pervasive nature of these systems can potentially fulfill the vision of a fully integrated digital and physical world. With collective adaptive systems, in the near future one may easily envision "enhanced" living and working environments, thanks to computing devices connected to every physical object that provide increasingly powerful capabilities of computing, storage of local data, communication with neighbours, physical sensing, and actuation. Such environments pave the way towards implementing any non-trivial pervasive computing service through the inherent distributed cooperation of a large set of devices, so as to address by self-adaptation the unforeseen changes in working conditions that necessarily happen—much in the same way adaptivity and resilience are addressed in complex natural systems at all levels, from molecules and cells to animals, species, and entire ecosystems [59].

A long-standing aim in computer science has indeed been to find effective engineering methods for exploiting mechanisms for adaptation and resilience in complex, large-scale applications. Practical adoption, however, poses serious challenges, since such mechanisms need to carefully trade efficiency for resilience, and are often difficult to predictably compose to meet more complex specifications. Despite much prior work, e.g., in macroprogramming, spatial computing, pattern languages, etc. (as surveyed in [8]), to date no such approach has provided a comprehensive workflow for efficient engineering of complex self-organising systems.

Recently, however, among the many related works (see Section 2), two key ingredients have been provided toward such an engineering workflow. First, the *computational field calculus* [21, 54] provides a language for specifying large-scale distributed computations and, critically, a functional programming model for their encapsulation and safely-scoped composition. This framework assumes that the system is composed of a discrete set of devices deployed in a space equipped with a notion of locality: each device works in asynchronous computational rounds producing a result data that is sent to local neighbours[1]. Second, a set of sufficient conditions for "self-stabilisation" have been identified [10, 19, 53], guaranteeing that a large class of programs are all self-adaptive systems resilient to changes in their environment—more precisely, after some period without changes in the computational environment, such a distributed computation reaches a stable state that only depends on inputs and network topology (i.e., the converged state is independent of computational history). As an example, such conditions reveal the non-resiliency of gossiping to find the minimum of a given value across a network: since each node continuously compute the minimum of all values received from neighbours, the system can't recover from the temporary decrease of a value below the minimum [19].

This paper combines these two advances with an approach to optimisation of self-organising systems via substitution of equivalent coordination mechanisms, guaranteed to result in the same functional behaviour though with different performance characteristics. Together, they combine into a workflow for efficient engineering of complex self-organising systems in which, once a distributed system is framed as a computation over fields, then: *(i)* a minimal resilient implementation is created, by composing building blocks from a library of reusable self-stabilising components or designed ad-hoc; *(ii)* performance is optimised by selective substitution of building block instances with alternate implementations, checking performance by simulation.

[1] Hence, we do not specifically deal with continuous functions and with virtual nodes that do not host computation—though they are mechanisms that might be mimicked: e.g., approximation of continuous functions can be developed along the lines of [11, 12].

This workflow is backed by pairing formal modelling and simulation of complex distributed systems. On the one hand, functional properties are addressed by a formally proved language of self-stabilising specifications, which also establishes functional equivalence of certain building blocks. On the other hand, dynamical properties are addressed by simulation, empirically evaluating performance differences when building blocks are selectively substituted by provably equivalent implementations. In particular, empirical analysis of large-scale systems, even though it may result in sub-optimality, is motivated by the fact that finding optimal combinations of alternative implementations easily becomes a computationally hard problem [22], that—to the best of our knowledge—has never been addressed.

The technical contributions of this paper with respect to previous work are: *(i)* building on [51], we provide the largest to date provably self-stabilising fragment of field calculus, by showing inevitable reachability of a unique stable state [19], including the self-organisation building blocks defined in [10]; *(ii)* we provide alternative implementations of these building blocks (some new and some consolidating existing algorithms), still in the self-stabilising fragment, and proved equivalent to the original versions; *(iii)* we empirically evaluate and compare performance of the building blocks and alternatives, characterising contexts in which a given implementation can be favoured.

The remainder of this paper is organised as follows: Section 2 reviews related work and discusses background and motivation, presenting the methodological workflow in the context of the field calculus; Section 3 formalises syntax, semantics and properties of the field calculus, providing building block examples showcasing its expressiveness; Section 4 presents our self-stabilisation framework, with formal definition and methodological implications; Section 5 provides the self-stabilising fragment, proof of self-stabilisation, proof of membership for the building blocks, and several motivating examples; Section 6 defines alternative building block implementations and empirically evaluates their performance; Section 7 presents two case studies illustrating the methodology; and Section 8 summarises and concludes.

2 RELATED WORK, BACKGROUND AND MOTIVATION

The approach we propose falls under the umbrella of *aggregate computing* [9], a framework for designing resilient distributed systems based on abstracting away from individual device behaviour: system design focusses instead on the aggregate behaviour of the collection of all (or a subset of) devices. In other words, aggregate computing considers the whole set of devices seen as a single "abstract computing machine." Coupled with a formal computational model, this approach aims at smooth composition of distributed behaviour, trading off expressiveness for control of system outcomes.

2.1 Relationship to Prior Work

Our work builds on two well-developed areas of prior work: aggregate programming languages, which address the challenges of programming collectives of devices, and self-stabilisation, which formalises a useful class of resilient system behaviours.

Aggregate programming. Aggregate programming methods of many sorts have been developed across a wide variety of applications and fields. A thorough review may be found in [8], which identifies four main approaches. First, "bottom-up" methods simplify aggregate programming by abstracting and simplifying programming of individual networked devices. These methods include TOTA [39], Hood [56], the chemical models by [55], Butera's "paintable computing" [13], and Meld [2]. In the context of parallel computing, the Bulk Synchronous Parallel (BSP) model [50] facilitates programming with barriers allowing multiple processors to synchronise, e.g., allowing system-wide computational rounds. Similarly, many cloud computing models (e.g., MapReduce [24]) provide bulk programming models that abstract away network structure.

Three families of "top-down" approaches complement these bottom-up methods. These higher-level approaches specify tasks for aggregates, then translate (e.g., by a compiler) from aggregate specifications into an implementation in terms of individual local actions. These approaches also tend to build in notions of implicit resilience, though specifics vary wildly from approach to approach. One such family focusses on spatial patterns, such as topological networks in Growing Point Language [16], geometric patterns in Origami Shape Language [42], self-healing geometries in [15] and [35], or universal patterns [57]. Another family instead aims at summarisation and streaming of information over regions of space and time. Examples include sensor-network query languages like TinyDB [38], Cougar [58], TinyLime [17], and Regiment [43].

The third family are general purpose space-time computing models. Some of these are spatial parallel computing models, such as StarLisp [36] and systolic computing (e.g., the works by [28] and [48]) that shift data in parallel on a structured network. Others, such as MGS [29, 30], are more topological in nature. Because of their generality, this class of computing models can form the basis of a layered approach to the construction of distributed adaptive systems, as in our previous work on field calculus [20, 21] and the generalised framework of aggregate programming [9, 51].

Self-stabilisation. This paper aims to find sufficient conditions identifying a large class of complex network computations with predictable outcomes despite transient changes in their environment or inputs, and to express this class by construction in terms of a language of resilient programs. The notion we focus on requires a unique global state (reached in finite time) independent of initial state, i.e., depending only on the environment (topology and sensors). We speak of this property as *self-stabilisation* as it is contained within the notion of self-stabilisation to *correct* states for distributed systems [26], defined in terms of a set C of correct states which the system enters in finite time and then never escapes from: in our case, C is the single state corresponding to the intended result obtained as a function from inputs and environment.

Several versions of self-stabilisation are found in literature, surveyed by [49], from works by [25] to more abstract ones [1], depending largely on the system model under study—protocols, state machines, and so on. In our case, self-stabilisation is studied for computational fields, considered as data structures distributed over space. However, since previous work trying to identify general conditions for self-stabilisation (e.g., by [32]) only considers very specific models (e.g., heap-like data structures in a non-distributed settings), it is difficult to make a precise connection with those prior results.

Some variations of the definition of self-stabilisation also deal with different levels of quality (e.g., fairness, performance). For instance, the notion of superstabilisation [27] extends the standard self-stabilisation definition by adding a requirement on a "passage predicate" that should hold while a system recovers from a specific topological change. Our work does not address this particular issue, since we completely equate treatment of topological changes and changes to inputs (e.g., sensors), and do not address specific performance requirements formally. Performance is also affected by the fairness assumption adopted: we relied on a notion abstracting from more concrete ones typically used [34]—these more concrete models could be applied with our work as well, but would reduce the generality of our results. Instead, we address performance issues in a rather different way: we allow for multiple different implementations of given building block functions, trading off reactiveness to different kinds of changes in different ways, proved equivalent in their final result, and selected based on empirical evaluation.

Concerning specific results on self-stabilisation, some approaches have achieved results that more closely relate to ours. [26] introduced a computation of minimum distance in hops from a source node that is known to self-stabilise and used it as a preliminary step in creating a graph spanning tree. Other authors attempt to devise general methodologies. [5] depict a compiler turning any protocol into a self-stabilising one. Though technically unrelated to our solution, it shares the philosophy of hiding details of how self-stabilisation is achieved under the hood of the execution platform: in our case designers are intended to focus on macro-level specification, trusting that components interact to achieve the global outcome in a self-stabilising way. Similarly, [31] suggest that hierarchical composition of self-stabilising programs is self-stabilising—a key idea for constructing our functional language of self-stabilising programs.

For this work's specific technical result in the context of the field calculus: apart from [51], which we extend here, the closest prior work appears to be [19] which, to the best of our knowledge, is the first attempt at directly connecting self-stabilisation to engineering self-organisation. In that work, self-stabilisation is proved for all fields inductively obtained by functional composition of fixed fields (sensors, values) and a spanning-tree-inspired spreading process. Here, we consider a more liberal programming language and also address dynamical properties by simulation. Finally, [37] develops an alternative approach to self-stabilisation for computational fields, using a fix-point semantics and currently including only structures based on spanning trees.

2.2 Computing with Fields

The basic data unit of aggregate computing is a dynamically changing *computational field* (or field for short) of values held across many devices. More precisely, a *field value* ϕ is a function $\phi : \mathbf{D} \to \mathcal{L}$ mapping each device $\delta \in \mathbf{D}$ to a local value $\ell \in \mathcal{L}$. Similarly, a *field evolution* is a dynamically changing field value, and a field computation takes field evolutions as input (e.g., from sensors or user inputs) and produces a field evolution as output, from which field values are (distributed) snapshots. For example, given a Boolean field input mapping certain devices to `True`, a `distanceTo` computation of an output field of estimated distances to the nearest such device can be constructed by iterative aggregation and spreading of information, with the output changing to track input changes. Note that while the computational field model maps most intuitively onto spatially-embedded systems, it can be used for any distributed computation (though it tends to be best suited for sparse networks).

Critical to the approach, any field computation can be mapped to an equivalent single device behaviour, to be iteratively executed by all devices in the network. Execution is in (per-device) *computation rounds*: sense-eval-broadcast iterations, in which a device collects information coming from neighbours and local sensors, the computation is evaluated against the device's local state, and a result of computation is broadcast to neighbours (which collect and use that state in their own future computation rounds).

2.3 Proposed Workflow

Our proposed workflow is based on computational field calculus [21] (or field calculus for short), a tiny functional language, in which any distributed computation can be expressed, encapsulated, and safely composed. Field calculus is a general-purpose language in which it is possible to express both resilient and non-resilient computations. For example, field calculus can express computing the minimum value in a network by gossip or by directed aggregation: the gossip implementation is non-resilient, because it cannot track a rising minimum, while the directed aggregation implementation is resilient and can track both rising and falling minimum values. Field calculus can, however, be restricted to a sub-language in which all programs are guaranteed resilient in the sense of self-stabilisation, as discussed in the following.

The succinctness of field calculus that makes formal proofs tractable, however, is not well suited for the practical engineering of self-organising systems, especially when one needs to scale to

$$\begin{array}{lcll}
\mathsf{P} & ::= & \overline{\mathsf{F}}\ \mathsf{e} & \text{program}\\
\mathsf{F} & ::= & \mathtt{def}\ \mathsf{d}(\overline{\mathsf{x}})\ \{\mathsf{e}\} & \text{function declaration}\\
\mathsf{e} & ::= & \mathsf{x} \mid \mathsf{v} \mid \mathtt{let}\ \mathsf{x} = \mathsf{e}\ \mathtt{in}\ \mathsf{e} \mid \mathsf{f}(\overline{\mathsf{e}}) & \text{expression}\\
 & & \quad \mid \mathtt{if}(\mathsf{e})\{\mathsf{e}\}\{\mathsf{e}\} \mid \mathtt{nbr}\{\mathsf{e}\} \mid \mathtt{rep}(\mathsf{e})\{(\mathsf{x}) \texttt{=>} \mathsf{e}\} & \\
\mathsf{v} & ::= & \ell \mid \phi & \text{value}\\
\ell & ::= & \mathsf{c}(\overline{\ell}) & \text{local value}\\
\phi & ::= & \overline{\delta} \mapsto \overline{\ell} & \text{neighbouring field value}\\
\mathsf{f} & ::= & \mathsf{d} \mid \mathsf{b} & \text{function name}
\end{array}$$

Fig. 1. Syntax of field calculus.

complex designs. This can be mitigated by reusable “building block” operators capturing common coordination patterns [10], thus raising the abstraction level and allowing programmers to work with general-purpose functionalities or user-friendly APIs capturing common use patterns.

These building blocks, despite desirable resilience properties, may not be particularly efficient or have desirable dynamical properties for a given application. We thus incorporate a new insight: due to the functional composition model and modular proof used in establishing the self-stabilising calculus, any coordination mechanism guaranteed to self-stabilise to the same result as a building block can be substituted without affecting the overall result. This allows alternative implementations in a “library of self-stabilising blocks,” functionally equivalent but trading off performance in different ways or with more desirable dynamics (e.g., specialised for particular applications) [3, 4].

Together, these insights enable a two-stage engineering workflow that progressively treats complex specification, resilience, and efficiency. The workflow starting point is specification of the aggregate behaviour to be implemented. Following this: *(i)* the specification is expressed as a composition of coordination patterns (e.g., information spreading, information collection, state tracking) that can be mapped onto building block operators, forming a “minimal resilient implementation” guaranteed self-stabilising but possibly far from optimal; *(ii)* each building block is then considered for replacement with a mechanism from the substitution library expected to provide better performance, confirming the improvement by analysis or simulation, then iterating, until a satisfactory level of performance is achieved. Finally, the library of building blocks can be naturally extended with new blocks and alternatives, as will likely be needed when addressing some novel application scenarios.

3 FIELD CALCULUS

This section presents first-order field calculus [20] with a syntax inspired by recent DSL implementations [14] (in place of the prior Scheme-like formulation in [20, 54]), then uses it to specify the key self-stabilising building blocks for this paper.

In our model, individual devices undergo computation in (local) asynchronous rounds: in each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbours while sleeping, performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation before going back to sleep. Our formulation assumes a denumerable set of device identifiers **D**, ranged over by δ, such that each device has a distinguished identifier. In the rest of the paper each device is represented by its identifier—our formalisation does not provide (and does not need) a syntax for devices.

3.1 Syntax

Figure 1 presents the syntax of field calculus. Following [33], overbar notation denotes metavariables over sequences and the empty sequence is •: e.g., for expressions, we let $\overline{\mathsf{e}}$ range over sequences of expressions $\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n$ ($n \geq 0$). Similarly, formulas with sequences are duplicated for each element of the sequences (assumed to be the same length): e.g., $\overline{\mathsf{f}(\mathsf{e}) = \mathsf{v}}$ is a shorthand for $\mathsf{f}_i(\mathsf{e}) = \mathsf{v}_i$ for $i = 1 \ldots |\mathsf{v}|$.

A program P consists of a sequence of function declarations and a main expression e. A function declaration F defines a (possibly recursive) function, with d the function name, $\overline{\mathsf{x}}$ the parameters and e the body. Expressions e model a whole field (i.e., e evaluates to a value on every device in the network, thus producing a computational field). As usual, the set of free variables in an expression e is denoted by **FV**(e), and we say an expression e is *closed* iff **FV**(e) is empty. An expression can be:

- a variable x, either a function formal parameter or local to a `let`- or `rep`-expression;
- a value, either a *local value* (associating each device to a computational value—e.g., numbers, literals—defined through data constructors c) or a *neighbouring field value* ϕ (associating each device to a map from neighbours to local values—note that such values appear in intermediate computations but not in source programs);
- a `let`-expression $\mathtt{let}\ \mathsf{x} = \mathsf{e}_0\ \mathtt{in}\ \mathsf{e}$, which is evaluated by first computing the value v_0 of e_0 and then yelding as result the value of the expression obtained from e by replacing all the occurrences of the variable x with the value v_0;
- a function call $\mathsf{f}(\overline{\mathsf{e}})$, where f can be either a *declared function* d or a *built-in function* b (such as accessing sensors, mathematical and logical operators, or data structure operations—see Electronic Appendix Afor examples);
- a conditional $\mathtt{if}(\mathsf{e}_1)\{\mathsf{e}_2\}\{\mathsf{e}_3\}$, splitting computation into two isolated sub-networks: devices evaluating e_1 to `True` compute expression e_2; the rest compute e_3;
- a `nbr`-expression $\mathtt{nbr}\{\mathsf{e}\}$, modelling neighbourhood interaction and producing a neighbouring field value ϕ that represents an "observation map" of neighbour's values for expression e, namely, associating each device to a map from neighbours to their latest evaluation of e;
- or a `rep`-expression $\mathtt{rep}(\mathsf{e}_1)\{(\mathsf{x}) \mathtt{=>} \mathsf{e}_2\}$, evolving a local state through time by evaluating an expression e_2, substituting variable x with the value calculated for the `rep`-expression at the previous computational round (in the first round x is substituted with the value of e_1). Although the calculus does not model anonymous functions, $(\mathsf{x}) \mathtt{=>} \mathsf{e}_2$ can be understood as an anonymous function with parameter x and body e_2.

Values associated to data constructors c of arity zero are written by omitting the empty parentheses, i.e., we write c instead of c(). We assume a constructor for each literal value (e.g., `False`, `True`, 0, 1, −1,...) and a built-in function bc for every data constructor c of arity $n \geq 1$, i.e., such that $\mathsf{bc}(\mathsf{e}_1, \ldots, \mathsf{e}_n)$ evaluates to $\mathsf{c}(\ell_1, \ldots, \ell_n)$ where each ℓ_i is the value of e_i. In case b is a binary built-in operator, we allow infix notation to enhance readability: i.e., we shall sometimes write $1 + 2$ for $+(1, 2)$. To simplify notation (and following features present in concrete implementations of field calculus [52], [47]), we shall also overload each (user-defined or built-in) function with local arguments to accept any combination of local and neighbouring field values: the intended meaning is then to apply the given function *pointwise* to its arguments. For example, let ϕ be the neighbouring field $\delta_1 \mapsto 1, \delta_2 \mapsto 2, \delta_3 \mapsto 3$ and ψ be $\delta_1 \mapsto 10, \delta_2 \mapsto 20, \delta_3 \mapsto 30$, we shall use $\phi + \psi$ for the pointwise sum of the two numerical fields giving the neighbouring field $\delta_1 \mapsto 11, \delta_2 \mapsto 22, \delta_3 \mapsto 33$, or $1 + \phi$ for the field obtained incrementing by 1 each value in ϕ, namely, $\delta_1 \mapsto 2, \delta_2 \mapsto 3, \delta_3 \mapsto 4$.

In the following we assume that the calculus is equipped with the type system defined by [20], which is variant of the Hindley-Milner type system [18] that has two kinds of types: local types (for local values) and field types (for neighbouring field values). This system associates to each local value a type L, and type field(L) to a neighbouring field of elements of type L, and correspondingly a type T to any expression.

As described in detail in Electronic Appendix B, to express more general and reusable functions, we add syntactic sugar to admit *functional parameters* in function definitions, written def d($\overline{\mathtt{x}}$)($\overline{\mathtt{z}}$){e} and called as follows d($\overline{\mathtt{e}}$)($\overline{\mathtt{f}}$), where the arguments $\overline{\mathtt{f}}$ can be either names of *plain* (i.e., non-extended) functions or functional parameters—names of extended functions are not allowed to be passed as arguments, and by convention, we omit the second parentheses whenever no functional parameters are present.

Example 3.1. As an example showcasing all classes of construct at work, consider the following definition of a distanceToWithObs function, mapping each device to an estimated distance to a source area (a numerical indicator field, holding 0 in the area and ∞ outside), computed as length of a minimum path that circumvents an obstacle area (a boolean indicator field):

```
def distanceTo(source)(metric) {
   rep (source) { (x) => min( source, minHood(nbr{x} + metric()) ) } }
def distanceToWithObs(source, obstacle)(metric) {
  if (obstacle) { infinity } { distanceTo(source)(metric) } }
```

In the body of function distanceToWithObs, construct if divides the space in two regions, where obstacle is True and where it is False: in the former the output is infinity, in the latter we compute—isolated from the devices in the former area, hence "circumventing it"—distance estimation by calling function distanceTo.

In the body of function distanceTo, we give 0 on sources through operator min on the indicator field source. On other devices, we compute the estimated distance as being infinity at the beginning, then evolving by taking the minimum value (minHood(field) is a built-in which returns the minimum value in field or ∞ if the field is empty) across neighbour estimates added pointwise to the estimated distance to each neighbor (obtained by functional parameter metric, to which built-in nbrRange can be passed that models a local range sensor).

3.2 Semantics

Operational semantics is formalized (in Electronic Appendix C): *(i)* for computation within a single device, by judgement "$\delta;\Theta \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta$", to be read "expression $\mathtt{e}_{\mathtt{main}}$ evaluates to θ on device δ with respect to environment Θ" (or "device δ fires"), where θ is an ordered tree of values tracking the results of all evaluated subexpressions of $\mathtt{e}_{\mathtt{main}}$, and Θ is a map from each neighbour device δ_i (including δ itself) to the θ_i produced in its last firing; *(ii)* for network evolution, by a transition system $\xrightarrow{act}$ on network configurations $N = \langle Env; \Psi \rangle$, where: Env models the environmental conditions (i.e., network topology and inputs of sensors on each device), Ψ models the overall status of the devices in the network at a given time (as a map from device identifiers to environments Θ), and actions *act* can either be firings of a device (δ) or network configuration changes (*env*).

3.3 Implementation of Building Blocks

We are now able to express the main "building blocks" used in field calculus (as reported in [10]), a set of highly general and guaranteed composable operators for the construction of resilient coordination applications. Each of these building blocks captures a family of frequently used strategies for achieving flexible and resilient decentralised behaviour, hiding the complexity of using the low-level constructs of field calculus. Despite their small number, these operators are so general as to cover, individually or in combination, a large number of the common coordination patterns used in design of resilient systems. The three building blocks, whose behaviour will be thoroughly evaluated in next section along that of alternative implementations, are defined as follows.

3.3.1 Block G. `G(source,initial)(metric,accumulate)` is a “spreading” operation generalising distance measurement, broadcast, and projection, which takes two fields and two functions as inputs: `source` (a float indicator field, which is 0 for sources and ∞ for other devices), `initial` (initial values for the output field), `metric` (a function providing a map from each neighbour to a distance), and `accumulate` (a commutative and associative two-input function over values). It may be thought of as executing two tasks: *(i)* computing a field of shortest-path distances from the source region according to the supplied function `metric`; and *(ii)* propagating values up the gradient of the distance field away from source, beginning with value `initial` and accumulating along the gradient with `accumulate`. This is accomplished through built-in `minHoodLoc`(ϕ, ℓ), which selects the minimum of the neighbours’ values in ϕ and the local value ℓ according to the lexicographical order on pairs.

```
def G(source, initial)(metric, accumulate) {
  rep ( pair(source, initial) ) { (x) =>
    minHoodLoc(pair(nbr{1st(x)}+metric(), accumulate(nbr{2nd(x)})), pair(source, initial))
} }
```

As an example, `G_distanceTo` function (equivalent to the function `distanceTo` shown in Section 3.1 with `metric` equal to `nbrRange`), and a `G_broadcast` function to spread values from a source, can be simply implemented with G as:

```
def addRange(x) { x + nbrRange() }
def identity(x) { x }
def G_distanceTo(source) { 2nd( G(source, 0)(nbrRange, addRange)) }
def G_broadcast(source, value) { 2nd( G(source, value)(nbrRange, identity)) }
```

3.3.2 Block C. `C(potential,local,null)(accumulate)` is an operation that is complementary to `G`: it accumulates information down the gradient of a supplied potential field. This operator takes three fields and a function as inputs: `potential` (a numerical field), `local` (values to be accumulated), `null` (an idempotent value for `accumulate`) and `accumulate` (a commutative and associative two-input function over values). At each device, the idempotent `null` is combined with the `local` value and any values from neighbours with higher values of the `potential` field, using function `accumulate` to produce a cumulative value at each device. For instance, if `potential` is a distance gradient computing with `G` in a given region R, `accumulate` is addition, and `null` is 0, then `C` collects the sum of values of `local` in region R.

```
def C(potential, local, null)(accumulate) {
  rep ( pair(local, uid()) ) { (x) =>
    pair(accumulate(
            mux(nbr{potential} < potential && nbr{2nd(x)} = uid(), nbr{1st(x)}, null),
            local  ),
         2nd(maxHood+(nbr{pair(potential, uid())})) )  }  }
```

As an example, a `C_sum` function summing all the values of a field down a potential, and a `C_any` function checking if any value of a boolean field is true and reporting the result down a potential, can be simply implemented with C as:

```
def sum_aux(field, local) { sumhood(field) + local }
def C_sum(potential, value) { 1st( C(potential, value, 0)(sum_aux)) }
def or_aux(field, local) { anyhood(field) || local }
def C_any(potential, value) { 1st( C(potential, value, false)(or_aux)) }
```

3.3.3 Block T. `T(initial,zero)(decay)` deals with time, whereas `G` and `C` deal with space. Since time is one-dimensional, however, there is no distinction between spreading and collecting, and thus only a single operator. This operator takes two fields and a function as inputs: `initial` (initial values for the resulting field), `zero` (corresponding final values), and `decay` (a one-input strictly decreasing function over values). Starting with `initial` at each node, that value gets decreased by function `decay` until eventually reaching the `zero` value, thus implementing a flexible count-down, where the rate of the count-down may change over time. For instance, if `initial` is a pair of a value `v` and a timeout t, `zero` is a pair of the blank value `null` and `0`, and `decay` takes a pair, removing the elapsed time since previous computation from the second component of the pair and turning the first component to `null` if the second reached `0`, then `T` implements a limited-time memory of `v`.

```
def T(initial, zero)(decay) {
  rep ( initial ) { (x) => min(max(decay(x), zero), initial) } }
```

As an example, a `T_track` function simply tracking an input value over time, and a `T_memory` function holding a value for a given amount of time (and then showing a null value), can be simply implemented with T as:

```
def T_track(value) { T(value, value)(identity) }
def memory_evolve(x) {
  if ( 1st(x) < sns_interval() ) { pair(0,null) } { pair(1st(x)-sns_interval(), 2nd(x)) } }
def T_memory(value, time, null) { 2nd(T(pair(time,value), pair(0,null))(memory_evolve)) }
```

with the built-in operator (sensor) `sns_interval` returning the time elapsed since the last execution round.

4 SELF-STABILISATION AND EVENTUAL BEHAVIOUR

In the dynamic environments typically considered by self-organising systems, a key resilience property is *self-stabilisation*: the ability of a system to recover from arbitrary changes in state. In particular, of the various notions of self-stabilisation (see the survey in [49]), we use the definition from [26] as further restricted by [19]: a self-stabilising computation is one that, from any initial state, after some period without changes in the computational environment, reaches a single "correct" final configuration, intended as the output of computation.

Self-stabilisation (formalised in Section 4.1) focusses on a computation's eventual behaviour (formalised in Section 4.2), rather than its transient behaviour, which also enables optimisation by substitution of alternate coordination mechanisms (cf. Section 2.3). As we will see, this definition covers a broad and useful class of self-organisation mechanisms, though some are excluded, such as continuously changing fields like self-synchronising pulse-coupled oscillators [40] and computations that converge only in the limit like Laplacian-based approximate consensus [44]. Incorporating such mechanisms into a framework such as we present here will require bounding the dynamical behaviours of computations (e.g., by identification of an appropriate Lyapunov function [23]). Preliminary investigations in this area have produced positive results (e.g., [23, 41]), but integration with the framework presented in this paper is a major project that remains as future work.

4.1 Self-Stabilisation

Our notion of self-stabilisation considers resilience to changes in the computational system's state or external environment. Hence, assume a program P and fixed environmental conditions *Env* (i.e., fixed network topology and inputs of sensors). According to the operational semantics outlined in Section 3.2, for each network configuration N with environment *Env* that is reachable from the empty network configuration, we can define a transition system $\langle \mathcal{S}, \xrightarrow{act} \rangle$ where:

- the only possible action labels *act* are device identifiers δ representing firings of an individual device of the network; and
- the set of the states $\mathcal{S}$ is the smallest set of the network configurations such that: (1) $N \in \mathcal{S}$, and (2) for each $N' \in \mathcal{S}$ and δ in the network there is an $N'' \in \mathcal{S}$ such that $N' \xrightarrow{\delta} N''$.

We say that a configuration N is *stable* iff it is not changed by firings, i.e., $N \xrightarrow{\delta} N$ for each δ. Let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$ be an infinite sequence of transitions in $\mathcal{S}$. We say that the sequence is *fair* iff each configuration N_t is followed by firings of every possible device, i.e., for each $t \geq 0$ and δ there exists a $t' > t$ such that $\delta_{t'} = \delta$. We say that the sequence stabilises to state N iff $N_i = N$ for each i after a certain $t \geq 0$.

Given a program P and fixed environmental conditions, a transition system like the one considered above can be defined for any closed expression e that may call the user-defined functions defined in P: just consider e as the main expression of P. In the following, for convenience of the presentation, we focus on computations associated to such an expression e.

Definition 1 (Stabilisation and Self-Stabilisation). *A closed expression* e *is: (i)* stabilising *iff every fair sequence stabilises given fixed environmental conditions Env; (ii)* self-stabilising *to state N iff every fair sequence stabilises to the same state N given fixed environmental conditions Env.*

A function f($\overline{\text{x}}$) *is* self-stabilising *iff given any self-stabilising expressions* $\overline{\text{e}}$ *of the type of the inputs of* f *the expression* f($\overline{\text{e}}$) *is self-stabilising.*

Note that if an expression e self-stabilises, then it does so to a state that is unequivocally determined by the environmental conditions *Env* (i.e., it does not depend on the initial configuration N_0) and can hence be interpreted as the output of a computation on *Env*. Furthermore, this final state N must be stable. Note that this definition implies that field computations recover from any change on environmental conditions, since they react to them by forgetting their current state and reaching the stable state implied by such a change. Complementarily, computation can generally reach a stable state only when environmental changes are transitory.

4.2 Eventual Behaviour

An environment *Env* is a pair $\langle \tau, \Sigma \rangle$, where: τ models *network topology* as a map from device identifiers to set of identifiers (representing a directed neighbouring graph); and Σ models *sensor (distributed) state* as a map from device identifiers to (local) sensors (i.e., sensor name/value maps denoted as σ). Define a *computational field* Φ as a map $\overline{\delta} \mapsto \overline{\text{v}}$,[2] such that if $\overline{\text{v}}$ have field type their domains are *coherent* with the environment $Env = \langle \tau, \Sigma \rangle$, that is, $\mathbf{dom}(\Phi(\delta)) = \tau(\delta) \cap \mathbf{dom}(\Phi)$. Let $\mathcal{V}[\![T]\!]$ be the set of values of type T and $\mathcal{T}[\![T]\!] = \mathbf{D} \twoheadrightarrow \mathcal{V}[\![T]\!]$ be[3] the set of all computational fields Φ of the same type. Each such Φ is computable by at least one self-stabilising expression e (defined

[2] Even though the definition resembles that of a *neighbouring field value*, it differs both in purpose and in content, since v is allowed to be a neighbouring field value itself, and $\overline{\delta}$ spans the whole network and not just a device's neighbourhood.

[3] By $A \twoheadrightarrow B$ we denote the set of all partial functions from A to B.

by cases, and executed in the restricted environment corresponding to $\mathbf{dom}(\Phi)$)—we say that $\mathtt{e}$ is a *self-stabilising expression for* Φ.

Note that a network status Ψ (see Section 3.2) induces uniquely a computational field Φ by defining $\Phi(\delta)$ as the root of the tree of values $\Psi(\delta)(\delta)$, while conversely each Φ is coherent with multiple network statuses Ψ. Thus a computational field is not sufficient to capture the whole status of a computation of a program P. However, for self-stabilising programs P and self-stabilising functions $\mathtt{f}$, it suffices to define the *eventual output* of a computation: given computational fields $\overline{\Phi}$, let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$ be any fair evolution of a network computing $\mathtt{f}(\overline{\mathtt{e}})$ where $\overline{\mathtt{e}}$ are self-stabilising expressions for $\overline{\Phi}$. Since $\mathtt{f}$ is self-stabilising, the fair evolution eventually stabilises to a uniquely determined state $N = \langle Env; \Psi \rangle$, independently from the chosen evolution and initial state. This final status field Ψ in turn determines a unique computational field Φ, which we can think of as the eventual output of the computation.[4]

Definition 2 (Eventual behaviour). *Let $\mathtt{e}$ be a self-stabilising closed expression. We write $[\![\mathtt{e}]\!]$ for the computational field Φ eventually produced by the computation of $\mathtt{e}$. Let $\mathtt{f}$ be a self-stabilising function of type $\overline{T} \rightarrow T'$, where $\overline{T} = T_1 \times \cdots \times T_n$ $(n \geq 0)$. We write $[\![\mathtt{f}]\!]$ for the mathematical function in $(\mathcal{T}[\![T_1]\!] \times \cdots \times \mathcal{T}[\![T_n]\!]) \rightarrow \mathcal{T}[\![T']\!]$,[5] such that $[\![\mathtt{f}]\!](\overline{\Phi}) = [\![\mathtt{f}(\overline{\mathtt{e}})]\!]$ where $\overline{\mathtt{e}}$ are self-stabilising expressions for $\overline{\Phi}$.*

Eventual behaviour provides a convenient viewpoint for compositional programming since, as shown by the next proposition (proved in Electronic Appendix D) it is preserved under substitutions.

Proposition 1 (Eventual behaviour preserving equivalences). (1) *Let $\mathtt{e}_1, \mathtt{e}_2$ be self-stabilising expressions with the same eventual behaviour. Then given a self-stabilising expression $\mathtt{e}$, swapping $\mathtt{e}_1$ for $\mathtt{e}_2$ in $\mathtt{e}$ does not change the eventual outcome of its computation.* (2) *Let $\mathtt{f}_1$, $\mathtt{f}_2$ be self-stabilising functions with the same eventual behaviour. Then given a self-stabilising expression $\mathtt{e}$, swapping $\mathtt{f}_1$ for $\mathtt{f}_2$ in $\mathtt{e}$ does not change the eventual outcome of its computation.* (3) *Let $\mathtt{e}$ be a self-stabilising expression calling a user-defined self-stabilising function $\mathtt{d}$ such that in body($\mathtt{f}$) no $\mathtt{x} \in$ args($\mathtt{f}$) occurs in the branch of an $\mathtt{if}$. Let $\mathtt{e}'$ be the expression obtained from $\mathtt{e}$ by substituting each function application of the kind $\mathtt{f}(\overline{\mathtt{e}})$ with body($\mathtt{f}$) $[$args$(\mathtt{f}) := \overline{\mathtt{e}}]$. Then $\mathtt{e}'$ is self-stabilising and has the same eventual behaviour as $\mathtt{e}$ (i.e. $[\![\mathtt{e}]\!] = [\![\mathtt{e}']\!]$).*

5 SELF-STABILISING FRAGMENT

By exploiting the definition of self-stabilisation given in previous section, and its implication in considering eventual behaviour as a valid characterisation of the functional property of a field computation, it is possible to identify sufficient conditions for self-stabilisation in terms of a fragment of the field calculus, inductively defined by: *(i)* identifying a "base" fragment of the field calculus that contains only self-stabilising programs; *(ii)* identifying a set of eventual behaviours preserving equivalences (cf. Proposition 1); and *(iii)* relying on the fact that the least fragment of the field calculus that contains the base fragment and is closed under the eventual behaviour preserving equivalences is self-stabilising.

Accordingly, in this section we first present some motivating examples of non self-stabilising field calculus programs (in Section 5.1), then present the syntax of the identified "base" self-stabilising fragment (in Section 5.2), then state the self-stabilisation result for the fragment along

[4] Note this eventual state is reached independently of the fair sequence of firing that occurs; hence, it would be the same also with firings following fully-synchronous concurrency models like BSP [50].

[5] Here we assume that all input computational fields share the same domain, which is to be intended as the domain of the overall computation.

$$\begin{array}{rcll}
\mathtt{s} & ::= & \mathtt{x} \mid \mathtt{v} \mid \mathtt{let}\ \mathtt{x} = \mathtt{s}\ \mathtt{in}\ \mathtt{s} \mid \mathtt{f}(\overline{\mathtt{s}}) \mid \mathtt{if(s)\{s\}\{s\}} \mid \mathtt{nbr\{s\}} & \text{self-stabilising expression}\\
& & \mid \mathtt{rep(e)\{(x)=>f^{C}(nbr\{x\}, nbr\{s\}, \overline{e})\}} & \\
& & \mid \mathtt{rep(e)\{(x)=>f(mux(nbrlt(s), nbr\{x\}, s), \overline{s})\}} & \\
& & \mid \mathtt{rep(e)\{(x)=>f^{R}(minHoodLoc(f^{MP}(nbr\{x\}, \overline{s}), s), x, \overline{e})\}} &
\end{array}$$

Fig. 2. Syntax of a self-stabilising fragment of field calculus expressions, where self-stabilising expressions s occurring inside a rep statement cannot contain free occurrences of the rep-bound variable x.

with equivalence results further extending the fragment (in Section 5.3), and finally discuss its expressiveness (in Section 5.4). The following examples will be discussed throughout this section:

```
def fcWrong(v) { rep (v) { (x) => v-x } }
def faWrong(v) { rep (v) { (x) => max(maxHood+(nbr{x}), v) } }
def fmWrong(v) { rep (v) { (x) => min(minHood(nbr{x}) - 1, v) } }
def fc(v)      { rep (v) { (x) => (v+x)/2 } }
def fa(v, p)   { rep (v) { (x) => max(maxHood+(mux(nbrlt(p), nbr{x}, 0), v) } }
def fm(v)      { rep (v) { (x) => min(minHood(nbr{x}) + 1, v) } }
```

5.1 Examples of non Self-stabilising Programs

Let us begin by considering some examples of field calculus programs that are not self-stabilising, illustrating key classes of program behavior that need to be excluded from our self-stabilising fragment—namely, oscillation, state preservation, and divergence.

Example 5.1. First, consider function fcWrong, it does not self-stabilise, since given a fixed input v its output loops through a series of different values. For example, if v is constantly equal to 1 the outputs are $0, 1, 0, 1, \ldots$ Thus in this case self-stabilisation is prevented by an *oscillating* behaviour.

Second, consider function faWrong (a classical gossip implementation): it does not self-stabilise, since its output depends on the whole history of values v given to it in the network. For example, if at some point a highest value k was given in some device, the eventual output of the function upon a fixed input $\mathtt{v} < k$ is k, thus it is not a function of the constant input v. Thus, in this case self-stabilisation is prevented by an indefinite "state preservation".

Finally, consider function fmWrong, with input v of an unbounded integer type (big integer): it does not self-stabilise, since given any fixed input v and at least one neighbour, its output keeps decreasing without a bound. Thus, in this case self-stabilisation is prevented by a *divergent* behaviour.

5.2 Syntax

The "base" self-stabilising fragment of field calculus is obtained by replacing each occurrence of the expression token e in the first two lines of Figure 1 (i.e., in the productions for P and F) with the self-stabilising expression token s, defined in Figure 2. This fragment includes: *(i)* all expressions not containing a rep construct, hence comprising built-in functions, which are therefore assumed to be self-stabilising; *(ii)* three special forms of rep-constructs, defined with a specific syntax coupled with semantic restrictions on relevant functional parameters.

5.2.1 The C, M, P, R *function properties.* The properties that these functional parameters are required to satisfy are among the following, visually annotated in the figure through superscripts on function names. Notice that properties M, P, and R require some of their argument types to be equipped with a *partial order* relation, while property C requires its argument types to be equipped with a *metric*. In order to obtain the self-stabilisation property for the fragment, we shall also need some further assumptions, discussed later in the description of each pattern.

C *(Converging).* A function $\mathtt{f}(\phi, \psi, \overline{\mathtt{v}})$ is said converging iff, for every device δ, its return value is closer to $\psi(\delta)$ than the maximal distance of ϕ to ψ. To be precise, given any environment Θ, device $\delta \in \mathbf{dom}(\Theta)$, values $\phi, \psi, \overline{\mathtt{v}}$ coherent with the domain of Θ, and assuming that $\delta; \Theta \vdash \mathtt{f}(\phi, \psi, \overline{\mathtt{v}}) \Downarrow \ell\langle\overline{\theta}\rangle$:

$$\text{dist}\,(\ell, \psi(\delta)) = 0 \text{ or } \text{dist}\,(\ell, \psi(\delta)) < \max\,\{\text{dist}(\phi(\delta'), \psi(\delta')) :\ \delta' \in \mathbf{dom}(\Theta)\}$$

where dist is any metric.

Example 5.2. Function $\mathtt{f}_1(\phi, \psi) = \mathtt{pickHood}(\psi - \phi) = (\psi - \phi)(\delta)$ is not converging, for example when ϕ, ψ are constant fields equal to $2, 3$ respectively so that $\ell = 1$ (pickHood selects the value on the current device from a field). On the other hand, functions $\mathtt{f}_2(\phi, \psi) = \mathtt{pickHood}((\psi + \phi)/2)$ and $\mathtt{f}_3(\phi, \psi) = \mathtt{pickHood}(\psi) + \mathtt{meanHood}(\phi - \psi)/2$ are converging.

M *(Monotonic non-decreasing).* A stateless[6] function $\mathtt{f}(\mathtt{x}, \overline{\mathtt{x}})$ with arguments of local type is monotonic non-decreasing in its first argument iff whenever $\ell_1 \leq \ell_2$, also $\mathtt{f}(\ell_1, \overline{\ell}) \leq \mathtt{f}(\ell_2, \overline{\ell})$.

Example 5.3. Function $\mathtt{f}_1(\ell) = \ell - 1$ is monotonic non-decreasing, while function $\mathtt{f}_2(\ell) = \ell^2$ is not.

P *(Progressive).* A stateless function $\mathtt{f}(\mathtt{x}, \overline{\mathtt{x}})$ with local arguments is progressive in its first argument iff $\mathtt{f}(\ell, \overline{\ell}) > \ell$ or $\mathtt{f}(\ell, \overline{\ell}) = \top$ (where $\top$ denotes the unique maximal element of the relevant type).

Example 5.4. Function $\mathtt{f}_1(\ell) = \ell + 1$ is progressive, while functions $\mathtt{f}_2(\ell) = \ell - 1$, $\mathtt{f}_3(\ell) = \ell^2$ are not.

R *(Raising).* A function $\mathtt{f}(\ell_1, \ell_2, \overline{\mathtt{v}})$ is raising with respect to partial orders $<, \lhd$ iff: *(i)* $\mathtt{f}(\ell, \ell, \overline{\mathtt{v}}) = \ell$; *(ii)* $\mathtt{f}(\ell_1, \ell_2, \overline{\mathtt{v}}) \geq \min(\ell_1, \ell_2)$; *(iii)* either $\mathtt{f}(\ell_1, \ell_2, \overline{\mathtt{v}}) \rhd \ell_2$ or $\mathtt{f}(\ell_1, \ell_2, \overline{\mathtt{v}}) = \ell_1$.

Example 5.5. Function $\mathtt{f}_1(\ell_1, \ell_2) = \ell_1$ is raising with respect to any partial orders. Function $\mathtt{f}_2(\ell_1, \ell_2) = \ell_1 - \ell_2$ is not raising since it violates both the first two clauses. Function $\mathtt{f}_3(\ell_1, \ell_2) = (\ell_1 + \ell_2)/2$ respects the first two clauses for $\lhd = <$, but it violates the last one whenever $\ell_2 > \ell_1$.

5.2.2 *The three* `rep` *patterns.* We are now able to analyse the three `rep` patterns.

Converging `rep`. In this pattern, variable `x` is repeatedly updated through function $\mathtt{f}^{\mathsf{C}}$ and a self-stabilising value `s`. The function $\mathtt{f}^{\mathsf{C}}$ may also have additional (not necessarily self-stabilising) inputs $\overline{\mathtt{e}}$. If the range of the metric granting convergence is a well-founded set[7] of real numbers, the pattern self-stabilises since it gradually approaches the value given by `s`.

Example 5.6. Function `fcWrong` does not respect the converging `rep` pattern, as shown in Example 5.2. However, if we change `fcWrong` to `fc` and assume that its input and output are finite-precision numeric values (e.g., Java's `double`), we obtain a *low-pass filter* that is self-stabilising and complies with the converging `rep` pattern.

[6] A function $\mathtt{f}(\overline{\mathtt{x}})$ is *stateless* iff given fixed inputs $\overline{\mathtt{v}}$ always produces the same output, independently from the environment or specific firing event. In other words, its behaviour corresponds to that of a mathematical function.

[7] An ordered set is *well-founded* iff it does not contain any infinite descending chain.

Acyclic `rep`. In this pattern, the neighbourhood's values for `x` are first filtered through a self-stabilising partially ordered "potential", keeping only values held in devices with lower potential (thus in particular discarding the device's own value of `x`). This is accomplished by the built-in function `nbrlt`, which returns a field of booleans selecting the neighbours with lower argument values, and could be defined as `def nbrlt(x) {nbr{x} < x}`.

The filtered values are then combined by a function `f` (possibly together with other values obtained from self-stabilising expressions) to form the new value for `x`. No semantic restrictions are posed in this pattern, and intuitively it self-stabilises since there are no cyclic dependencies between devices.

Example 5.7. Function `faWrong` does not respect the acyclic `rep` pattern, since it aggregates all neighbours without any "acyclic filtering". However, if we change `faWrong` to `fa` we obtain a particular usage of the C block, which is self-stabilising and complies with the acyclic `rep` pattern.

Minimising `rep`. In this pattern, the neighbourhood's values for `x` are first increased by a monotonic progressive function $\mathtt{f}^{\mathsf{MP}}$ (possibly depending also on other self-stabilising inputs). As specified above, $\mathtt{f}^{\mathsf{MP}}$ needs to operate on local values: in this pattern it is therefore implicitly promoted to operate (pointwise) on fields.

Afterwards, the minimum among those values and a local self-stabilising value is then selected by function $\mathtt{minHoodLoc}(\phi, \ell)$ (which selects the "minimum" in $\phi[\delta \mapsto \ell]$). In order to be able to define such a minimum, we thus require the partial order $\leq$ to constitute a *lower semilattice*.[8]

Finally, this minimum is fed to the *raising* function $\mathtt{f}^{\mathsf{R}}$ together with the old value for `x` (and possibly any other inputs $\overline{\mathtt{e}}$), obtaining a result that is higher than at least one of the two parameters. We assume that the second partial order $\lhd$ is *noetherian*,[9] so that the raising function is required to eventually conform to the given minimum.

Intuitively, this pattern self-stabilises since it computes the minimum among the local values ℓ after being increased by $\mathtt{f}^{\mathsf{MP}}$ along every possible path (and the effect of the raising function can be proved to be negligible).

Example 5.8. Function `fmWrong` does not respect the minimising `rep` pattern, since its internal function is monotonic (see Example 5.3) but not progressive (see Example 5.4). However, if we change `fmWrong` to `fm` we obtain a *hop-count distance*, a particular instance of the G block which is self-stabilising and complies with the minimising `rep` pattern.

Note that the well-foundedness and noetherianity properties are trivially verified whenever the underlying data set is finite.

5.3 Self-Stabilisation and Equivalence

Under reasonable conditions, we are able to prove that the proposed fragment is indeed self-stabilising. The proofs of all the results in this section are given in the Electronic Appendix E, while here we only report the full statements.

Theorem 1 (Fragment Stabilisation). *Let* `s` *be a closed expression in the self-stabilising fragment, and assume that every built-in operator is self-stabilising.*[10] *Then* `s` *is self-stabilising.*

[8] A *lower semilattice* is a partial order such that greatest lower bounds are defined for any finite set of values in the partial order. In the examples used in this paper we shall treat *greatest lower bounds* as *minima*, since the only examples of such partial orders we consider are in fact total orders.

[9] A partial order is *noetherian* iff it does not contain any infinite ascending chains.

[10] Most built-in operators are stateless, thus trivially self-stabilising in one round.

Since the fragment is closed under function application, the result is immediately extended to whole programs.

In Section 4.2 we introduced a notion of *equivalence* for self-stabilising programs. Therefore, although the `rep` patterns are defined through *functions* with certain properties, we are allowed to inline them (which is a transformation preserving self-stabilisation, as shown in Proposition 1). Moreover, a few noteworthy equivalence properties hold for the given patterns, as shown by the following theorem.

Theorem 2 (Substitutability). *The following three equivalences hold: (i) each* `rep` *in a self-stabilising fragment self-stabilises to the same value under arbitrary substitution of the initial condition; (ii) the* converging `rep` *pattern self-stabilises to the same value as the single expression* `s` *occurring in it; (iii) the* minimising `rep` *pattern self-stabilises to the same value as the analogous pattern where* $\mathtt{f}^R$ *is the identity on its first argument.*

In other words, the function $\mathtt{f}^R$ does not influence the eventual behaviour of a function, and can instead be used to fine-tune the transient behaviour of an algorithm. The same holds for the initial conditions of all patterns and function $\mathtt{f}^C$ in the converging `rep` pattern (which in fact is only meant to fine-tune the transient behaviour of the given expression `s`). No relevant equivalences can be stated for the acyclic `rep` pattern, since it is parametrised by a single aggregating function which in general heavily influences the final outcome of the computation.

5.4 Expressiveness

5.4.1 Programs captured by the fragment. Even though at a first glance the fragment could seem rather specific, it encompasses (equivalent versions of) many relevant algorithms. In particular, all of the three building blocks introduced in Section 3.3 are easily shown to belong to the fragment. This effectively constitutes a new and simpler proof of self-stabilisation for them.

Operator *G* is the following instance of the minimising `rep` pattern:

```
def fr(new, old) { new }
def fmp(field, dist)(accumulate) {
  pair(1st(field) + dist, accumulate(2nd(field))) }
def G(source, initial)(metric, accumulate) {
  rep(pair(source, initial)){ (x) =>
    fr(minHoodLoc(fmp(nbr{x}, metric())(accumulate), pair(source, initial)), x) } }
```

Function `fr` is trivially raising (with respect to any pair of partial orders), and function `fmp` is monotonic progressive provided that pairs are ordered lexicographically (since `dist` is a positive field).

Operator *C* is the following instance of the acyclic `rep` pattern:

```
def f(field, local, null, potential)(accumulate) {
  pair(accumulate(mux(2nd(field) = uid(), 1st(field), null), local),
    2nd(maxHood+(nbr{pair(potential, uid())})) ) }
def C(potential, local, null)(accumulate) {
  rep(pair(local, uid())){ (x) =>
    f(mux(nbrlt(potential), nbr{x}, null), local, null, potential)(accumulate) } }
```

Operator *T* is the following instance of the converging `rep` pattern:

```
def fc(cur, lim, initial)(decay) {
  min(max(decay(pickHood(cur)), pickHood(lim)), initial) }
def T(initial, zero)(decay) {
  rep(initial){ (x) => fc(nbr{x}, nbr{zero}, initial)(decay) } }
```

Function `fc` is converging since `decay(pickHood(cur))` is granted to be closer to `zero` than its argument, hence:

$$|\mathtt{fc}(\phi, \mathtt{nbr}\{\mathtt{zero}\}, \mathtt{v}) - \mathtt{zero}| < |\phi(\delta) - \mathtt{zero}| \leq \max(|\phi - \mathtt{nbr}\{\mathtt{zero}\}|)$$

Furthermore, the present fragment strictly includes the one defined in [51]. Both fragments include all expressions without the `rep` construct. The first and third `rep` pattern in [51] are special cases of *converging* `rep` (the first converges to $\mathtt{v}_0$ in the *bounded* condition and the third to ℓ in the *double bounded* condition). The second pattern is almost exactly equivalent to the *acyclic* `rep`.

In the following Section 6 we shall show further examples of algorithms still belonging to the fragment, which are alternative implementations of G, C and T.

5.4.2 Programs not captured by the fragment. Unfortunately, many self-stabilising programs are not captured by the fragment. In most cases this is due to syntactical reasons, so that the critical program P can in fact be rewritten into an equivalent program P', which instead belongs to the fragment. An example of this issue is given by the three building blocks G, C and T, which we needed to rewrite in order to make them fit inside the self-stabilising fragment (see Section 5.4.1).

Furthermore, self-stabilising programs exist which cannot be rewritten to fit inside the fragment. As an example, one such program is the *replicated gossip* [45] algorithm, which does not fit inside the fragment. In particular, replicated gossip is "self-stabilising" *provided* that a certain parameter p (refresh period) is set to a large enough value with respect to certain network characteristics—and as such, it would require a slight modification of our definition of self-stabilisation as well.

6 ALTERNATIVE BUILDING BLOCKS

Even though the G, C, and T building blocks define a useful and versatile base of operators, in practice better performing alternatives are often preferred in some specific conditions (see for example the work in [51]). We can also use the fragment itself to get inspiration for new alternatives or interesting variations of existing ones. Importantly, the self-stabilisation framework allows alternatives to be assessed on empirical grounds even when the dynamics of their operation are imperfectly understood, allowing engineering decisions to be made even when analytical solutions are not available.

In the exploration to follow, we compare the performance of each operator and an alternative via simulation. We evaluate each proposed alternative by simulating a network of 100 devices placed uniformly randomly in a $200m \times 20m$ rectangular arena, with a $30m$ communication radius. The dynamics of self-stabilisation are studied by introducing perturbations in "space" or "time". In the space perturbation experiments, devices run asynchronously at 1 Hz frequency, moving at 1 m/s in a direction randomly chosen at every round. We shall consider "small spatial perturbation" where this is the entirety of the perturbation, and "large spatial perturbation" where the source for the spreading / aggregation of the information also switches from the original device to an alternate device every 200 seconds. On the other hand, in the "time perturbation", devices remain still, but their operating frequency is randomly chosen between 0.9 Hz and 1.1 Hz (small perturbation) or 0.5 Hz and 2 Hz (large perturbation). We performed 200 simulations per configuration, letting both the control and alternate building blocks run at the same time. Experiments are performed using the Alchemist simulator [46].[11]

[11] For the sake of reproducibility, the actual experiments are made available at: https://bitbucket.org/danysk/experiment-2017-tomacs

6.1 Alternative G

The G operator can be understood as the computation of a distance measure w.r.t. a given metric, while also propagating values according to an accumulating function. However, naive computation of distance suffers from the *rising value problem*: the rising rate of distance values is bounded by the shortest distance in the network, possibly enforcing a very slow convergence rate. Some algorithms avoiding this problem have been developed, such as the CRF-gradient algorithm [7]. It is possible to rewrite a CRF-gradient distance calculuation to fit the present fragment, as in the following (adapted from the code implemented in the Protelis library [47]):

```
def raise(new, old, speed, dist) {
  let constraint = minHood(nbr{1st(old)} + dist + (nbrLag()+sns_interval())*2nd(old)) in
  if (new = old || 1st(new) = 0 || constraint <= 1st(old)) {  new } {
    pair(1st(old)+speed, speed/sns_interval())  } }
def combine(x, dist) {  pair(1st(x) + dist, 0) }
def CRF(source, speed)(metric) {
  rep ( pair(source, 0) ) { (x) =>
    raise(minHoodLoc(combine(nbr{x}, metric()), pair(source, 0)), x, speed) } }
```

where `nbrLag` returns a field of communication lags from neighbours.

It is easy to see that `raise` is raising with respect to the two identical partial orders $\leq$, $\leq$ (the output either increases the `old` value or conforms to the `new` value). Notice that this rewriting effectively constitutes an alternative proof of self-stabilisation for the algorithm.

If it is acceptable to lose some degree of accuracy, another possibility for avoiding the rising value problem is to introduce a *distortion* into the metric. This is the approach chosen by the Flex-Gradient algorithm [6] (which we will abbreviate FLEX). This algorithm allows for a better response to transitory changes while reducing the amount of communication needed between devices. In this case also, we can equivalently rewrite the algorithm in order to make it fit into the self-stabilising fragment.

```
def raise(new, old, dist, eps, freq, rad) {
  let slopeinfo = maxHood(triple((1st(old) - nbr{1st(old)})/dist, nbr{1st(old)}, dist)) in
  if (new = old || 1st(new) = 0 || 2nd(old) = freq || 1st(old) > max(2*1st(new), rad)) { new } {
    if (1st(slopeinfo) > 1+eps) { pair(2nd(slopeinfo) + (1+eps)*3rd(slopeinfo), 2nd(old)+1) } {
      if (1st(slopeinfo) < 1-eps)) {  pair(2nd(slopeinfo) + (1-eps)*3rd(slopeinfo), 2nd(old)+1)   } {
        pair(1st(old), 2nd(old)+1)   } } } }
def combine(x, dist) { pair(1st(x) + dist, 0) }
def FLEX(source, epsilon, frequency, distortion, radius)(metric) {
  rep ( pair(source, 0) ) { (x) =>
    let dist = max(metric(), distortion*radius) in
    raise(minHoodLoc(combine(nbr{x}, dist), pair(source, 0)), x, dist, epsilon, frequency, radius) } }
```

In this case, `raise` is raising with respect to the two partial orders $\leq_1$ (ordering w.r.t. the first component of the pair) and $\leq_2$ (ordering w.r.t. the second component).

We evaluate these new building blocks when applied to distance estimation, using the two following variations of `G_distance` (parameter `r` in the body of `G'_flex_distance` stands for the communication radius of devices):

```
def G'_crf_distance(source) {  CRF(source, 1/12)(nbrRange) }
def G'_flex_distance(source) {  FLEX(source, 0.3, 10, 0.2, r)(nbrRange) }
```

Figure 3 shows the evaluation of G and its proposed replacements: FLEX has a good performance all-around, while CRF suffers poor performance with small spatial disruptions and G suffers poor performance with large spatial disruptions.

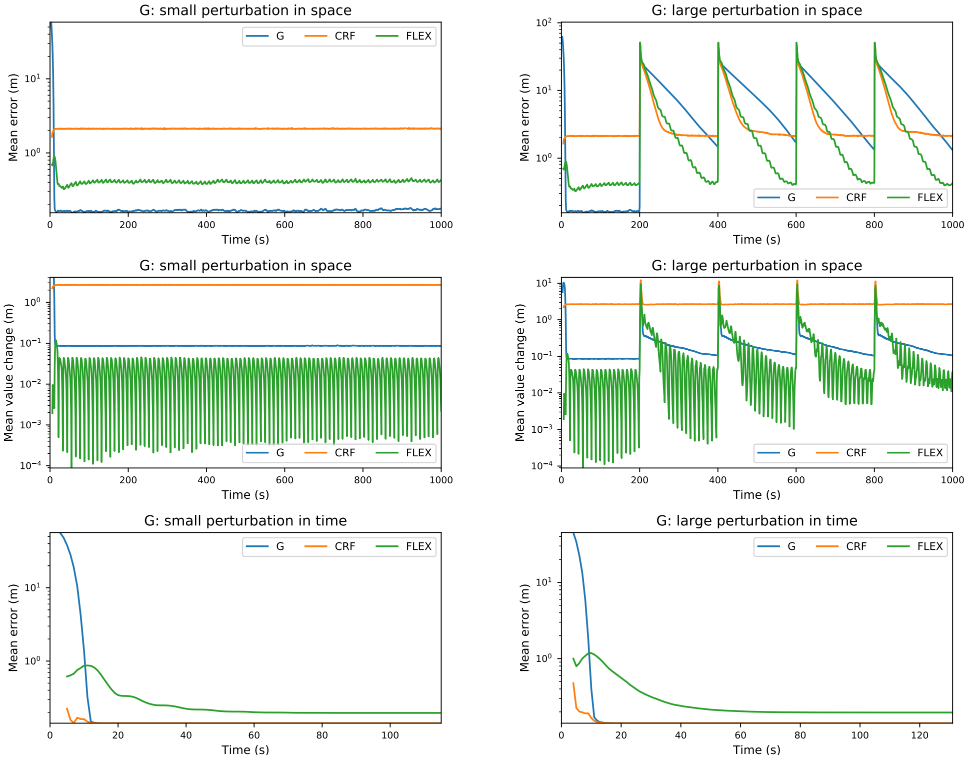


Fig. 3. Evaluation of G building blocks: plain G (blue), CRF (green) and FLEX (red). We measure the average error across all devices (first and last row) and the stability of the value, namely, the average value change between subsequent rounds (middle row). With small spatial perturbations, G provides the lowest average error, while FLEX provides the highest local value stability. With large spatial changes, CRF is the quickest to adapt, but stabilises with a higher error than FLEX. The classic G suffers from the rising value problem. All the algorithms stabilise in time with little sensitivity to device asynchrony.

6.2 Alternative C

The C operator aggregates a computational field of `local` values with the function `accumulate` towards the device with highest potential, each device feeding its value to the neighbour with highest potential. This process, however, is fragile since the "neighbour with highest potential" changes often and abruptly over time. In order to overcome this shortcomings, it is sometimes possible to use a *multipath C*.

Assume that the aggregating operator defines an abelian monoid[12] on its domain. Assume in addition that each ℓ in the domain has an n-th root ℓ_n, that is, an element which aggregated with itself n times produces the original value ℓ. Then the value computed by a device can be "split" and sent to *every* neighbour device with higher potential than the current device, by taking its n-th root where n is the number of devices with higher potential.

```
def extract(val, num)(root) { pair(val, root(val, num)) }
def aggregate(field, local, potential)(accumulate, root) {
  extract( accumulate(foldHood(2nd(field), accumulate), local),
    counthood(nbr{potential} > potential) )(root) }
```

[12] A structure $\langle X, \circ \rangle$ is an abelian monoid if $\circ$ is an associative and commutative operator with identity.

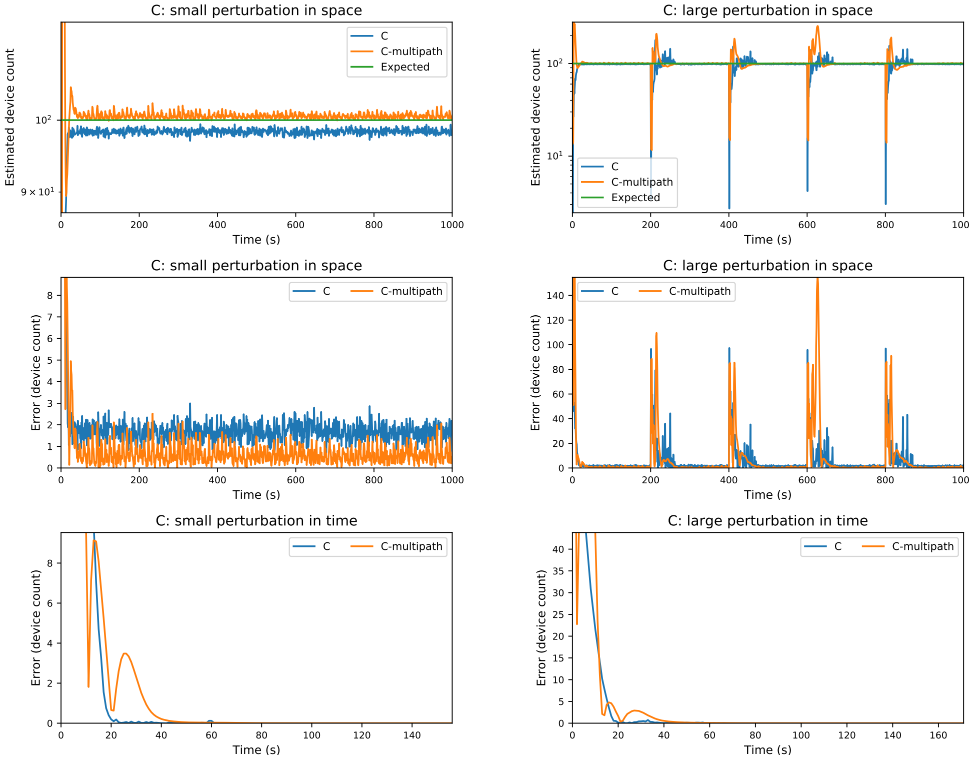


Fig. 4. Evaluation of C building blocks: classic C (blue), and multi-path alternative (green). Expected values are depicted in red. We measure the aggregated value in the source node (first row) and the error (last two rows). With small spatial perturbations, the multipath alternative outperforms the spanning-tree-building default implementation; however, it may provide worse estimations at the beginning of transients that require a large reconfiguration. Both algorithms stabilise regardless of devices' asynchrony.

```
def C'(potential, local, null)(accumulate, root) {
  rep ( pair(local,local) ) { (x) =>
    aggregate(mux(nbr{potential} < potential, nbr{x}, null),
      local, potential)(accumulate, root) } }
// C'_sum application
def C'_sum(potential, field) {  1st( C'(potential, value, 0)(+, /)) }
```

We evaluate the multi-path alternative of C when used to sum values of a field, using the `C'_sum` variation of `C_sum`:[13] Specifically, we compare `C_sum` and `C'_sum` used to aggregate the summation of "1" along the gradient of a distance estimate produced by the FLEX algorithm. As a consequence, we expect to get the count of devices participating to the system in the source of the distance estimate. Since the source switches in case of large perturbation, the counting device switches as well. Figure 4 shows the evaluation of C and its proposed replacement: the multi-path version performs better with small spatial changes, but may return higher errors during transients that require a whole network reconfiguration.

[13] Operator / is used as root for C' since a value gets equally divided by n and spread in the n neighbour nodes ascending potential.

6.3 Alternative T

Both the T operator and the whole *converging* `rep` pattern are meant to smooth out the outcome of another computation, which at the limit is returned unaltered. However, it is sometimes useful to introduce a *spatial* coordination among different devices, in order to smooth out the converging process also spatially. This can be accomplished by the following alternative building block, which *decays* towards a *value* with a speed obtained by *averaging* on how close each neighbour is to its goal value.

```
def follow(cur, lim)(average, decay) {  pickHood(lim) + decay(average(cur - lim)) }
def T'(initial, value)(average, decay) {
  rep ( initial ) { (x) => follow(nbr{x}, nbr{value})(average, decay) } }
// T'_track application
def T'_track(value) {  T'(value,value)(meanHood, x => a*x)}
```

We evaluate the use of T’ in tracking a noisy signal, using `T’_track` variation of `T_track` where `meanHood` computes the mean value of the provided field, and `a` is the smoothing parameter. In the comparison of `T_track` and `T’_track`, every device perceives the original signal (either a sine or a square wave) summed with a locally generated noise in $[-1, 1]$[10] (s). In particular, `T’_track` provides a sort of spatial low-pass filter, that trades a delay in tracking the signal for a smoother response. Figure 5 aggregates the results. T’ takes advantage of the spatial smoothing, and performs better overall in case of noisy input. This comes, however, at the price of lower reactivity to changes, which becomes evident with large enough values of the smoothing parameter.

7 APPLICATION EXAMPLES

We now illustrate, with two application examples, how distributed applications can be implemented on top of the proposed building blocks (hiding the low-level coordination mechanisms `rep` and `nbr`), and then quickly adjusted and optimised toward specific performance goals by switching the specific building block implementation that is used, using the variants presented in previous section. Both of the scenarios that we consider are in a pervasive computing environment, and focus on a network of personal devices (e.g., phones, smart watches) spread through an urban environment. In these scenarios, devices move with the person carrying them along the walkable areas of the city, and can only indirectly influence movement (e.g., by presenting a message to their user).

For the first scenario, we consider a community festival, with acts performing in various venues, and wish to track the number of people watching each act over time. Here, we will consider a person to be watching an act if they are part of a continuous region of crowd that is closer to that act than to any other act. This computation can be implemented by using G to partition the space into zones of influence, by means of a potential field of which each act is a source (as in function `distanceTo`). We then use C to sum a field counting the number of people closely surrounded by others, and thus forming a crowd (as in function `summarise`). Finally, T is used for smoothing both the crowd estimates and the results over time. The resulting code, expressed using the functions described in previous section, is as follows:

```
def crowdSize(acts, crowd) { T_track(C_sum(G_distanceTo(acts), T_track(crowd))) }
```

To test this example application in simulation, we distributed a network of 300 devices randomly distributed across the city centre of the Italian city of Cesena. In this simulation, pedestrians walk at 1.4 meters per second from their initial position towards an act randomly chosen between the two located in distinct large spaces of the city (Piazza del Popolo and Giardini Savelli), as depicted in Figure 6. Devices run asynchronously, performing a round of computation and communication every five seconds, and communicating by broadcast within a radius of 150 metres (ignoring buildings

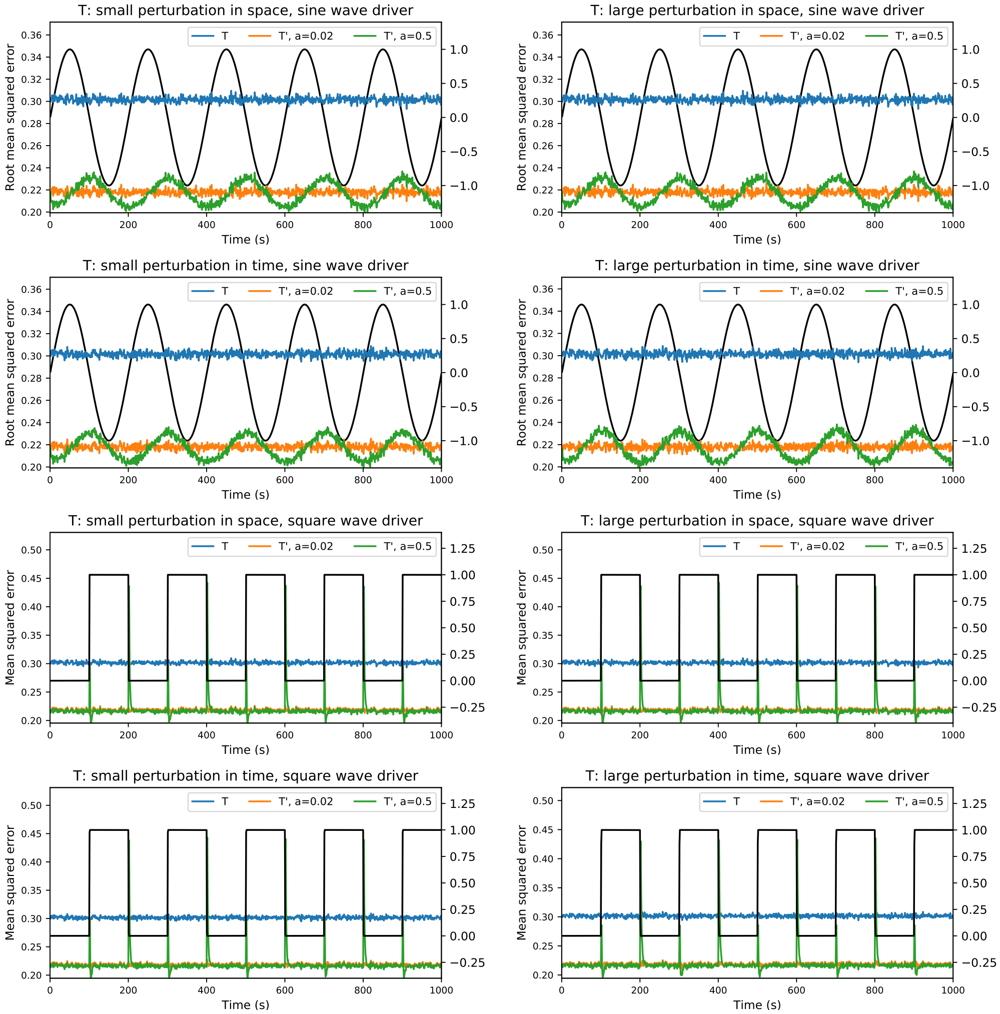


Fig. 5. Evaluation of T (blue) and T' building blocks with different smoothing parameter values (a = 0.02 in green, and a = 0.5 in red). The driver signal (plotted in black for reference) is locally summed with a random noise in $[-1, 1]^{10}$ and fed to the algorithm for tracking. We measure the root mean squared error in the devices' response for small (left column) or large (right column) perturbations in either space (first and third row) or time (second and last row). T' outperforms T in every scenario but the square wave transient: the smoothing with the neighbouring devices, in fact, greatly mitigates the local introduction of noise at the price of a lower reactivity to signal changes. The smoothing parameter can be interpreted as controlling a trade-off between such reactivity and the smoothness of the response. In our testbed, T' shows minimal sensibility to any kind of perturbation.

and other physical obstacles). Our implementation is realised in Protelis [47] and simulations were performed using Alchemist [46]. We note that Alchemist is a generalised GIS framework for multi-agent simulations, not a specialised crowd simulator, but higher-fidelity crowd simulations are not necessary for studying the adaptation dynamics of the information system.

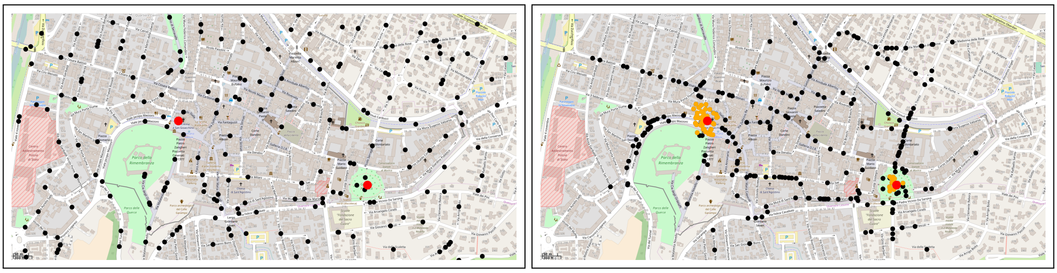
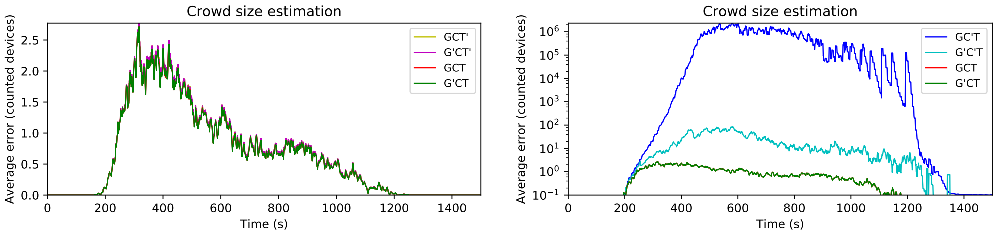


Fig. 6. Screenshots from simulation of crowd size estimation scenario: acts are indicated as red dots, pedestrians are black, and pedestrians who are part of a contiguous crowd are orange. From their initial position, people walk towards an act of interest following the pedestrian roads, becoming counted as part of a crowd once they have clumped up close to an act.



(a) G’ improves over G

(b) C’ fails, but is mitigated by G’

Fig. 7. Key results for the crowd size estimation scenario: a) Use of G’ slightly improves performance over G, while T performs slightly better than T’. b) The C’ algorithm fails badly due to the network being both sparse and volatile, mandating preference of C in this case. The problems with C’ can be largely mitigated by substitution of G’ instead of G, though the choice of T versus T’ does not have any significant effect.

In this scenario, we execute eight variants of the `crowdSize` algorithm, all combinations of the building blocks and alternates developed in the previous section: G or G’ (FLEX), C or C’ (multipath), and T or T’. We measure the error for each combination as the absolute value of the difference between estimated and true counts for people watching each act, namely,

$$\frac{1}{|A|}\sum_{a \in A} |\hat{P_a} - P_a|$$

where A is the set of acts a, $|A|$ is the number of acts, $\hat{P_a}$ is the estimated count of people watching act a as computed by the algorithm, and P_a is the true count of people watching an act.

Figure 7 presents key results, averaged over 51 simulation runs. In these simulations, adopting G’ instead of G produces a slight improvement in performance. On the other hand, it turns out that C’ fails badly, always making the results much worse, likely due to the combination of both the high volatility of the network and the sparsity induced by city streets. This failure, however, can be mitigated by applying G’, which produces a potential function that is much more stable in response to large perturbations. The choice of T versus T’ has much less impact: T’ performs slightly worse than T in combination with C’ and does not mitigate the failure of C’.

The second example considers signaling an evacuation alert signal to a pre-defined zone, along with the proposal of a suggested evacuation path. This is implemented using T to track whether

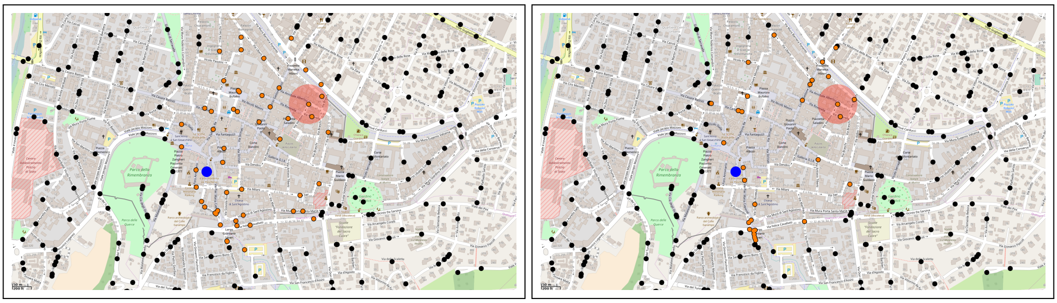


Fig. 8. Screenshots from simulation of evacuation alert scenario: devices are initially randomly scattered through the city centre (black dots). After alert (translucent red circle) is enabled, and devices in the evacuation zone are signaled (orange) by the action of the coordinator (blue) and begin trying to leave the zone.

any device in the zone is currently alerted (using G to create a potential field to a static device selected as coordinator, and C to perform a logical or as in function any), then using G to broadcast that value from the coordinator throughout the zone and again to compute paths to the non-alerted areas outside of the zone. Finally, the mux operator is used to differentiate computations on devices inside and outside of the alert zone.

```
def evacuationAlert(zone, coordinator, alert) {
  G_distanceTo( mux(zone, false,
      G_broadcast(coordinator, T_track(C_any(G_distance(coordinator), alert))))) }
```

Simulations for this experiment used the same environment of 300 devices spread through the center of Cesena, with the same model of asynchronous execution and communication, the only difference being that devices perform a round of computation and communication every two seconds rather than every five seconds. In this simulation, devices are initially stationary, and the alert signal is enabled starting at time $t = 20$ seconds of simulated time from the start of the simulation. Since devices are unable to directly affect the movement of the people holding them, however, we simulate the people acting on the alert not by following the direction provided by any of the simulated algorithms, but walking toward the closest waypoint outside of the evacuation zone. Such behaviour is depicted in Figure 8.

As before, we execute eight variants, covering all combinations of the three building blocks and their alternates. We measure the error for each algorithm as the mean of the minimum mean square error between the angles of the suggested evacuation vector and the optimal one for each node, normalised in $[0, 1]$, with the special rule that devices that are in alert zone when they shouldn't be or not in the alert zone when they should be get the maximum error, namely:

$$\text{error} = \frac{1}{N} \sum_{d \in D} \begin{cases} 0 & \text{not in zone and not alerted} \\ \left(\frac{\min(|\alpha_d - \hat{\alpha_d}|, 2\pi - |\alpha_d - \hat{\alpha_d}|)}{\pi}\right)^2 & \text{in zone and alerted} \\ 1 & \text{otherwise (alert/zone mis-match)} \end{cases}$$

where N is the number of devices initially inside the zone, D is the collection of all devices, $\hat{\alpha_d}$ is the computed direction (angle) for device d, and α_d is its actual ideal direction. The minimum function is used in order to always pick the smallest angle between the two separating the optimal vector and the suggested one (namely, the difference of the two and 2π minus that value). This outputs an error in the $[0, \pi]$ range, that we normalise linearly into $[0, 1]$.

In this scenario, we find that two of the proposed alternative implementations of the self-stabilising building blocks significantly improve performance. Figure 9 shows the results, averaged

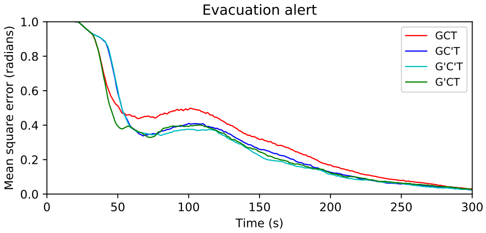


Fig. 9. Evacuation alert scenario results: G' and C' each improve performance significantly over G and C, respectively, and using both improves it incrementally further. Choice of T versus T' has no significant effect.

over 51 simulation runs. G', in particular, performs from equivalently to much better than G along the whole simulated time span. The behaviour of C' is more complex: it has a longer reaction time as compared to C, as it is more sensitive to large perturbations. As soon as the initial transient phase is over, however, C' provides a consistent improvement over the performance of the original C implementation. Using C' and G' together provides a further (though smaller) performance increment. The choice of T' versus T, however, has no significant impact on performance.

Together, these results illustrate how our approach enables fast, lightweight implementation and optimisation of distributed applications. Different applications are best served by different combinations and tradeoffs in the dynamics of building block implementations: for example, G' improved over G in both scenarios, while C' help the second but not the first, and neither had noisy enough changes for T' to significantly improve on T. The approach we have implemented allows such combinations to be rapidly and safely explored, enabling optimisation of distributed systems without their re-design.

8 CONCLUSIONS

Using computational field calculus as "lingua franca" for an abstract, uniform description of self-organising computations, we have identified a large class of self-stabilising distributed algorithms, including general "building block" operators that simplify the specification of programs within this class. The class is formalised as a fragment of the field calculus, closed under composition, and flexible enough to also include various alternative building block implementations, allowing dynamical performance optimisation with guaranteed convergence to the same values. This self-stabilising fragment is at the core of a methodology for efficient engineering of self-organising systems, rooted in modelling and simulation: *(i)* a system specification is constructed using formally-proved self-stabilising building blocks, and *(ii)* alternative implementations of building blocks are switched in selected points of the specification to improve performance, with performance improvement detected by empirical means such as simulations.

An important future direction is more detailed characterisation for the dynamic trade-space, to enable more systematic optimisation via mechanism substitution. In addition to making human engineering easier, this may also enable automated optimisation, both during engineering and dynamically at run-time. Alternative definitions of *self-stabilisation* may allow capture and description of wider classes of resilient program behaviours (e.g., replicated gossip [45]) or better modeling of important aspects of spatial computations (e.g., space-time information), as well as integration with dynamical response models such as those in [23, 41]. Other potential improvements include expansion of the library of building blocks (including to non-spatial systems), identification of more substitution relationships between building blocks and high-performance resilient coordination mechanisms, and development and deployment of applications based on this approach.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for insightful comments and suggestions. This work has been partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 644298 HyVar (Damiani), ICT COST Action IC1402 ARVI (Damiani), Ateneo/CSP project RunVar (Damiani), and the United States Air Force and the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750- 10-C-0242 (Beal) and Contract No. HR001117C0049 (Beal). The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations. Approved for public release, distribution unlimited.

REFERENCES

[1] Anish Arora and Mohamed G. Gouda. 1993. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering* 19 (1993), 1015–1027.

[2] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. 2007. Meld: A Declarative Approach to Programming Ensembles. In *IEEE International Conference on Intelligent Robots and Systems (IROS '07)*. 2794–2800.

[3] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. 2017. Compositional Blocks for Optimal Self-Healing Gradients. In *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society, 91–100.

[4] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. 2017. Optimally-Self-Healing Distributed Gradient Structures Through Bounded Information Speed. In *Coordination Models and Languages*. Lecture Notes in Computer Science, Vol. 10319. Springer, 59–77.

[5] Baruch Awerbuch and George Varghese. 1991. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *IEEE Symp. on Foundations of Comp. Sci. (FOCS91)*. 258–267.

[6] Jacob Beal. 2009. Flexible Self-healing Gradients. In *ACM Symposium on Applied Computing (SAC '09)*. ACM, 1197–1201.

[7] Jacob Beal, Jonathan Bachrach, Daniel Vickery, and Mark Tobenkin. 2008. Fast self-healing gradients. In *ACM Symposium on Applied Computing (SAC'08)*. ACM, 1969–1975.

[8] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. 2013. Organizing the Aggregate: Languages for Spatial Computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, Marjan Mernik (Ed.). IGI Global, Chapter 16, 436–501.

[9] Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate Programming for the Internet of Things. *IEEE Computer* 48, 9 (2015).

[10] Jacob Beal and Mirko Viroli. 2014. Building Blocks for Aggregate Programming of Self-Organising Applications. In *8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. IEEE Computer Society, 8–13.

[11] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. 2016. Self-Adaptation to Device Distribution Changes. In *10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society, 60–69.

[12] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. 2017. Self-Adaptation to Device Distribution in the Internet of Things. *TAAS* 12, 3 (2017), 12:1–12:29.

[13] William Butera. 2002. *Programming a Paintable Computer*. Ph.D. Dissertation. MIT, Cambridge, USA.

[14] Roberto Casadei and Mirko Viroli. 2016. Towards Aggregate Programming in Scala. In *First Workshop on Programming Models and Languages for Distributed Computing (PMLDC '16)*. Article 5, 7 pages.

[15] Lauren Clement and Radhika Nagpal. 2003. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*.

[16] Daniel Coore. 1999. *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. Ph.D. Dissertation. MIT, Cambridge, MA, USA.

[17] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. 2005. Mobile data collection in sensor networks: The TinyLime middleware. *Elsevier Pervasive and Mobile Computing Journal* 4 (2005), 446–469.

[18] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Symposium on Principles of Programming Languages*. ACM, 207–212.

[19] Ferruccio Damiani and Mirko Viroli. 2015. Type-based Self-stabilisation for Computational Fields. *Logical Methods in Computer Science* 11, 4 (2015).

[20] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. 2016. A type-sound calculus of computational fields. *Science of Computer Programming* 117 (2016), 17 – 44.

[21] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. 2015. Code Mobility Meets Self-organisation: A Higher-Order Calculus of Computational Fields. In *Formal Techniques for Distributed Objects, Components, and Systems*. Lecture Notes in Computer Science, Vol. 9039. Springer, 113–128.

[22] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. 2013. Synthesis of fixed-point programs. In *Int.l Conf. on Embedded Software, EMSOFT 2013*. IEEE, 22:1–22:10.

[23] Soura Dasgupta and Jacob Beal. 2016. A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*. IEEE, 7282–7287.

[24] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[25] EW Dijkstra. 1982. EWD391 Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 41–46. EWD391's original date is 1973.

[26] Shlomi Dolev. 2000. *Self-Stabilization*. MIT Press.

[27] Shlomi Dolev and Ted Herman. 1997. Superstabilizing Protocols for Dynamic Distributed Systems. *Chicago Journal of Theoretical Computer Science* (1997).

[28] Bradley R. Engstrom and Peter R. Cappello. 1989. The SDEF programming system. *J. Parallel and Distrib. Comput.* 7, 2 (1989), 201 – 231.

[29] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. 2002. *Computational models for integrative and developmental biology*. Technical Report 72-2002. UniversitÃľ d'Evry, LaMI.

[30] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. 2005. Computations in Space and Space in Computations. In *Unconventional Programming Paradigms*. Lecture Notes in Computer Science, Vol. 3566. Springer, Berlin, 137–152.

[31] Mohamed G. Gouda and Ted Herman. 1991. Adaptive programming. *IEEE Transactions on Software Engineering* 17 (1991), 911–921.

[32] Ted Herman and Imran Pirwani. 2001. A composite stabilizing data structure. In *International Workshop on Self-Stabilizing Systems*. Lecture Notes in Computer Science, Vol. 2194. Springer, 167–182.

[33] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001).

[34] Mehmet H. Karaata and Pranay Chaudhuri. 1998. A self-stabilizing algorithm for strong fairness. *Computing* 60 (1998), 217–228.

[35] Attila Kondacs. 2003. Biologically-inspired Self-Assembly of 2D Shapes, Using Global-to-local Compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

[36] C. Lasser, J.P. Massar, J. Miney, and L. Dayton. 1988. *Starlisp Reference Manual*. Thinking Machines Corporation.

[37] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. 2017. Asynchronous Distributed Execution Of Fixpoint-Based Computational Fields. *Logical Methods in Computer Science* 13, 1 (2017).

[38] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2002. TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 131–146.

[39] Marco Mamei and Franco Zambonelli. 2009. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. on Software Engineering Methodologies* 18, 4 (2009), 1–56.

[40] Renato E. Mirollo and Steven H. Strogatz. 1990. Synchronization of pulse-coupled biological oscillators. *SIAM J. Appl. Math.* 50, 6 (1990), 1645–1662.

[41] Yuanqiu Mo, Jacob Beal, and Soura Dasgupta. 2017. Error in Self-Stabilizing Spanning-Tree Estimation of Collective State. In *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. IEEE Computer Society.

[42] Radhika Nagpal. 2001. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. Ph.D. Dissertation. MIT, Cambridge, MA, USA.

[43] Ryan Newton and Matt Welsh. 2004. Region Streams: Functional Macroprogramming for Sensor Networks. In *Workshop on Data Management for Sensor Networks*. 78–87.

[44] R. Olfati-Saber, J. A. Fax, and R. M. Murray. 2007. Consensus and Cooperation in Networked Multi-Agent Systems. *Proc. IEEE* 95, 1 (January 2007), 215–233.

[45] Danilo Pianini, Jacob Beal, and Mirko Viroli. 2016. Improving Gossip Dynamics Through Overlapping Replicates. In *Coordination Models and Languages*. Lecture Notes in Computer Science, Vol. 9686. Springer, 192–207.

[46] Danilo Pianini, Sara Montagna, and Mirko Viroli. 2013. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation* 7, 3 (2013), 202–215.

[47] Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: Practical Aggregate Programming. In *ACM Symposium on Applied Computing 2015*. 1846–1853.

[48] F Raimbault and D Lavenier. 1993. ReLaCS for Systolic Programming. In *International Conference on Application-Specific Array Processors*. 132–135.

[49] Marco Schneider. 1993. Self-stabilization. *Comput. Surveys* 25 (1993), 45–67.

[50] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[51] Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 2015. Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields. In *9th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society, 81–90.

[52] Mirko Viroli, Jacob Beal, and Kyle Usbeck. 2013. Operational Semantics of Proto. *Science of Computer Programming* 78, 6 (June 2013), 633–656.

[53] Mirko Viroli and Ferruccio Damiani. 2014. A Calculus of Self-stabilising Computational Fields. In *Coordination Languages and Models*. Lecture Notes in Computer Science, Vol. 8459. Springer-Verlag, 163–178.

[54] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. 2013. A Calculus of Computational Fields. In *Advances in Service-Oriented and Cloud Computing*. Communications in Computer and Information Science, Vol. 393. Springer Berlin Heidelberg, 114–128.

[55] Mirko Viroli, Danilo Pianini, Sara Montagna, Graeme Stevenson, and Franco Zambonelli. 2015. A coordination model of pervasive service ecosystems. *Science of Computer Programming* 110 (2015), 3 – 22.

[56] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. 2004. Hood: a neighborhood abstraction for sensor networks. In *2nd Int.l Conf. on Mobile systems, applications, and services*. ACM Press.

[57] Daniel Yamins. 2007. *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*. Ph.D. Dissertation. Harvard, Cambridge, MA, USA.

[58] Yong Yao and Johannes Gehrke. 2002. The Cougar Approach to In-Network Query Processing in Sensor Networks. *SIGMOD Record* 31 (2002), 2002.

[59] Franco Zambonelli and Mirko Viroli. 2011. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications* (2011).

Received ??; revised ??; accepted January 2018