Engineering Resilient Collective Adaptive Systems by Self-Stabilisation

Mirko Viroli, Giorgio Audrito, Jacob Beal,
Ferruccio Damiani, Danilo Pianini

Abstract

Collective adaptive systems are an emerging class of networked computational systems, particularly suited in 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.

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

1

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 [57].

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 [6]), 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* [52, 19] 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 [51, 8, 17], 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 exposes that minimum of the values received from neighbours, the system can't recover from the temporary decrease of a value below the minimum [17].

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:

1. a minimal resilient implementation is created, by composition of building blocks extracted from a library of reusable self-stabilising components or designed ad-hoc;

2. optimisation of performance is achieved by selective substitution of building block instances with alternate implementations, whose performance is checked by simulation.

This workflow is backed by pairing formal modelling and simulation of com-

---

[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 [10].

plex distributed systems. On the one hand, functional properties are addressed by building systems on top of a formally proved language of self-stabilising specifications, which also establishes the functional equivalence of certain building blocks. On the other hand, dynamical properties are addressed by simulation, empirically evaluating the different performance of systems in which building blocks are selectively substituted by provably equivalent implementations. In particular, the use of empirical analysis for the large-scale systems we consider (even though it may result in sub-optimality), is motivated by the fact that finding an optimal combination of alternative distributed systems implementations easily becomes a computationally hard problem [20], that – to the best of our knowledge – has never been addressed in literature.

The technical contributions of this paper with respect to previous work are the following:

- we provide a simplified operational semantics of the first-order field calculus, reducing the formalisation in [18] along the lines of the approach proposed in [19] for the higher-order version of the calculus;

- by developing on [49], we provide the largest to date fragment of the calculus that is provably self-stabilising, using a proof methodology showing inevitable reachability of a unique stable state [17]—the calculus is shown to include the basic self-organisation building blocks defined in [8];

- 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;

- we empirically evaluate and compare the performance of the alternate versions of the building blocks, characterising the 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 the background and motivation for this paper, presenting the methodological workflow in the context of the field calculus; Section 3 then 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 implementations of the building blocks, 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 in this paper falls under the umbrella of *aggregate computing* [7], a framework for designing resilient distributed systems based on the idea of abstracting away from the behaviour of individual devices: system design focusses instead on the global, aggregate behaviour of the collection of all (or a subset of) the devices. Put in other words, aggregate computing considers as "abstract computing machine" the whole set of devices seen as a single

"body." Coupled with a formal computational model, this approach has the goal of providing smooth composition of distributed behaviour, and trading off expressiveness with the ability to control the outcome of system design. This is done by relying on a formal model guaranteeing functional properties and relying on other means (such as simulation) for assessing dynamical properties.

2.1 Relationship to Prior Work

The work presented in this paper 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 [6], in which four main approaches to aggregate programming are identified. First, many "bottom-up" methods aim to simplify aggregate programming by focusing on abstracting and simplifying the programming of individual networked devices. These methods include TOTA [37], the Hood sensor network abstraction [54], the chemical models by [53], Butera's "paintable computing" hardware model [11], and Meld [2]. In the context of parallel computing, the Bulk Synchronous Parallel (BSP) model [48] facilitates programming by enacting synchronisation barriers that allow multiple processors to synchronise, e.g., up to allow for computation to proceed on system-wide rounds. Similarly, a number of cloud computing models (e.g., MapReduce [22]) provide bulk programming models that abstract away network structure completely or nearly so.

Three families of "top-down" approaches are complementary to these bottom-up methods. These higher-level approaches specify tasks for aggregates and then translate, by means of compilers or similar software, from aggregate specifications into a set of individual local actions that can implement the desired aggregate behaviour. These approaches also tend to build in at least some notion of implicit resilience, though the specifics vary wildly from approach to approach.

One of these families focusses on creating geometric and topological patterns, such as the topological networks of Growing Point Language [14], the geometric patterns of Origami Shape Language [40], the self-healing geometries by [13] and [33], or Yamins' universal patterns [55]. Another largely disjoint family instead aims at summarisation and streaming of information over regions of space and time. Examples include sensor-network query languages like TinyDB [36], Cougar [56], TinyLime [15], and Regiment [41].

The third family, generalising over all of the prior approaches, are general purpose space-time computing models. Some of these are spatial parallel computing models, such as StarLisp [34] and systolic computing (e.g., the works by [26] and [46]), which use parallel shifting of data on a structured network. Others, such as MGS [27, 28], 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 [18, 19] and the generalised framework of aggregate programming [7, 49].

Self-stabilisation

The primary focus of the work in this paper is to find sufficient conditions for identifying a large class of complex network computations whose outcome is predictable despite transient changes in their environment or inputs, and to express this class in terms of a language of resilient programs that can be used to create such systems by construction. The notion we focus on requires a unique global state (being reached in finite time) depending only on the state of the environment (topology and sensors), that is, independent of the initial state. We speak of this property as *self-stabilisation* since it is contained within the notion of self-stabilisation to *correct* states for distributed systems [24], defined in terms of a set C of correct states into which the system eventually enters in finite time, and then never escapes from—in our case, C is made up of only the single state eventually reached and corresponding to the intended result of computation, obtained as a function from inputs and environment.

Several versions of the notion of self-stabilisation can be found in literature, surveyed by [47], from works by [23] to more abstract ones [1], usually depending on the reference model for the system to study—protocols, state machines, and so on. In our case, self-stabilisation is studied for computational fields, which can be considered as data structures distributed over space. However, since previous work trying to identify general conditions for self-stabilisation (e.g., by [30]) 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 super-stabilisation [25] 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 the treatment of topological changes and changes to the 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 [32]—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. [24] introduced a hop-count gradient (computing minimum hop-by-hop distance from a source node) that is known to self-stabilise and used it as a preliminary step in the creation of the spanning tree of a graph. Other authors attempt to devise general methodologies. [3] depicted a compiler turning any protocol into a self-stabilising one. Though this is technically unrelated to our solution, it shares the philosophy of hiding the details of how self-stabilisation is achieved under the hood of the execution

platform: in our case in fact, the designer is intended to focus on the macro-level specification, trusting that components behave and interact so as to achieve the global outcome in a self-stabilising way. Similarly, [29] suggested that hierarchical composition of self-stabilising programs is self-stabilising—an idea that is key here to construct a whole functional language of self-stabilising programs.

Concerning the specific technical result achieved here in the context of the field calculus, and apart form the work by [49] that we extend here, the closest prior work appears to be the work by [17] which, to the best of out knowledge, is the first attempt at providing a notion of self-stabilisation directly connected to the problem of engineering self-organisation. As in the present work, self-stabilisation is not proved for a specific algorithm or system, but is proved for all fields inductively obtained by functional composition of fixed fields (sensors, values) and by a spanning-tree-inspired spreading process. In this paper we consider a more liberal programming language and also address dynamical properties by simulation. Finally, an alternative approach to prove self-stabilisation for computational fields is developed in [35], in which it is seen in terms of a fix-point semantics, and currently includes only structures based on spanning trees.

2.2   Computing with Fields

The computational model of aggregate computing uses as basic unit of data a dynamically changing *computational field* (or field for short) of values held across many devices in the network. More precisely, a *field value* ϕ is a function $\phi : D \to \mathcal{L}$ that maps each device δ in domain D to a local value ℓ in range $\mathcal{L}$. Similarly, a *field evolution* is a dynamically changing field value, and a field computation can be seen as taking field evolutions as input (e.g., from sensors or user inputs) and producing a field evolution as output, from which field values are (distributed) snapshots. For example, given an input of a Boolean field mapping certain devices of interest to *true*, an output field of estimated distances to the nearest such device can be constructed by iterative aggregation and spreading of information, such that as the input changes the output changes to match—this computation is referred to in this paper as `distanceTo`; it is also sometimes elsewhere referred to as a *gradient computation* (e.g., [5, 4]). 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, of which spatially-embedded systems are an example).

Critically to the approach, any field computation can be properly turned into an equivalent single device behaviour, to be iteratively executed by all devices in the network. Namely, this is carried on in (per-device) *computation rounds*: sense-eval-broadcast iterations, in which information coming from neighbours and from local sensors are collected in a device, the computation is evaluated against the device's local state, and a result of computation is broadcast to neighbours (which will collect and use that state in their own future rounds of computation).

2.3 Proposed workflow

Our proposed workflow is based on computational field calculus [19] (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 complex designs. This can be mitigated by highly reusable "building block" operators capturing common coordination patterns [8], 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 their desirable resilience properties, may not be particularly efficient or have desirable dynamical properties in the specific application at hand. 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 that is guaranteed to self-stabilise to the same result as a building block can be substituted for that building block without affecting the self-stabilisation of the overall program, including its final output. This allows us to include alternative implementations in our "library of self-stabilising blocks:" blocks that are functionally equivalent but trade off performance in different ways or have more desirable dynamics (typically specialised for particular applications of the building blocks, as the base operators are extremely generic).

Together, these insights enable a two-stage engineering workflow that progressively treats complex specification, resilience, and efficiency. The starting point for the workflow is a specification of the desired aggregate behaviour to be implemented by the self-organising system. Following this:

1. 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. The result is a "minimal resilient implementation" guaranteed to be self-stabilising but possibly far from optimal.

2. 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 satisfying level of performance is achieved.

Finally, given the intrinsic extensibility of our approach, our library of building blocks can be naturally extended with new blocks and/or alternative block

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

Figure 1: Syntax of field calculus.

implementations, as will likely be needed when addressing some novel application scenarios.

3   Field Calculus

In this section, we present the first-order Field Calculus [18], with a syntax inspired by recent DSLs implementing the constructs of the calculus [12] (in Section 3.1),[2] its operational semantics (in Section 3.2), a convenient minimal extension allowing for functional parameters (in Section 3.3) and examples including key building blocks for the paper (in Section 3.4).

Our formulation assumes a denumerable set of device identifiers, 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 the field calculus. Following [31], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by •. E.g., for expressions, we let $\overline{\texttt{e}}$ range over sequences of expressions, written $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n$ $(n \geq 0)$. Similarly, formulas containing one or more sequences in overbar notation are supposed to be duplicated for each element of the sequences (which are assumed to share the same length): e.g., $\overline{\texttt{f}}(\texttt{e}) = \overline{\texttt{v}}$ is a shorthand for $\texttt{f}_i(\texttt{e}) = \texttt{v}_i$ for $i = 1 \ldots |\texttt{v}|$.

A program $\texttt{P}$ consists of a sequence of function declarations and of a main expression $\texttt{e}$. A function declaration $\texttt{F}$ defines a (possibly recursive) function, where $\texttt{d}$ is the function name, $\overline{\texttt{x}}$ are the parameters and $\texttt{e}$ is the body. Expressions $\texttt{e}$ are the main entities of the calculus, modelling a whole field (that is, an expression $\texttt{e}$ evaluates to a value on every device in the network, thus producing a computational field). An expression can be:

- a variable $\texttt{x}$, declared either as function formal parameter or as local to a let- or rep-expression;

---

[2]The original formulation of the Field Calculus [52, 18] uses a Scheme-like syntax reflecting earlier implementations [50].

- a value, which in turn could be either a *local value* (associating each device to a computational value – numbers, literals, and so on – 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 are allowed to appear in intermediate computations but not in source programs);

- a `let`-expression `let x = e`$_0$ `in 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 `f(`$\bar{\text{e}}$`)`, 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);

- a conditional `if(e`$_1$`){e`$_2$`}{e`$_3$`}`, which is evaluated by splitting the computation into two sub-networks working in isolation: the devices that evaluated e_1 to `True` altogether compute expression e_2, the devices that evaluated e_1 to `False` compute e_3;

- a `nbr`-expression `nbr{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 `rep(e`$_1$`){(x)=>e`$_2$`}`, evolving a local state through time by evaluating an expression e_2, substituting the variable `x` with the value calculated for the whole `rep`-expression at the previous computational round (in the first computation round `x` is substituted with the value of e_1). Although the calculus does not model anonymous functions, the syntax `(x)=>e`$_2$ can be understood as defining an anonymous function with parameter `x` and body e_2.

The set of free variables in an expression `e` is denoted by $\mathbf{FV}(e)$. As usual, we say that an expression `e` is *closed* iff $\mathbf{FV}(e)$ is empty.

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 `bc(e`$_1$`,...,e`$_n$`)` evaluates to `c(`$\ell_1, ..., \ell_n$`)` 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 [50], [45]), 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, $\delta_1 \mapsto 2, \delta_2 \mapsto 3, \delta_3 \mapsto 4$.

Built-in Function	Type Signature	Meaning
$\texttt{uid}()$	$() \to \texttt{num}$	*device identifier*
$\texttt{+},\texttt{-},\texttt{*},\texttt{/}$	$(\texttt{num},\texttt{num}) \to \texttt{num}$	*arithmetical operators*
$\texttt{<},\texttt{<=},\texttt{=},\texttt{>=},\texttt{>}$	$(\texttt{num},\texttt{num}) \to \texttt{bool}$	*comparison operators*
$\texttt{\&\&},\texttt{||}$	$(\texttt{bool},\texttt{bool}) \to \texttt{bool}$	*boolean operators*
$\texttt{mux}(\texttt{b},\ell,\ell)$	$\forall t.(\texttt{bool},t,t) \to t$	*multiplex selection*
$\texttt{pair}(\ell,\ell)$	$\forall t_1 t_2.(t_1,t_2) \to \texttt{tuple}(t_1,t_2)$	*pair construction*
$[\,\overline{\ell}\,]$	$\forall \overline{t}.(\overline{t}) \to \texttt{tuple}(\overline{t})$	*tuple construction*
$\texttt{1st}(\ell),\texttt{2nd}(\ell),\texttt{3rd}(\ell)$	$\forall \overline{t}.(\texttt{tuple}(\overline{t})) \to t_i \ (i=1,2,3)$	*tuple element access*
$\texttt{pickHood}(\phi)$	$\forall t.(\texttt{field}(t)) \to t$	*value in current device*
$\texttt{foldHood}(\phi,\ell)(\texttt{f})$	$\forall t.(\texttt{field}(t),t,(t,t) \to t) \to t$	*general neighbour aggregation*
$\texttt{meanHood}(\phi)$	$\forall t.(\texttt{field}(t)) \to t$	*average of neighbour values*
$\texttt{maxHood}(\phi),\texttt{maxHood+}(\phi)$	$\forall t.(\texttt{field}(t)) \to t$	*maximum of neighbour values*
$\texttt{minHood}(\phi),\texttt{minHood+}(\phi)$	$\forall t.(\texttt{field}(t)) \to t$	*minimum of neighbour values*
$\texttt{minHoodLoc}(\phi,\ell)$	$\forall t.(\texttt{field}(t),t) \to t$	*minimum of neighbor & local values*
$\texttt{nbrRange}(),\texttt{nbrLag}()$	$() \to \texttt{field}(\texttt{num})$	*space-time distance from neighbours*
$\texttt{snsNum}()$	$() \to \texttt{num}$	*generic numeric sensor*
$\texttt{sns\_interval}()$	$() \to \texttt{num}$	*interval with previous round*

Figure 2: Built-in functions used throughout this paper, with types and meaning.

In the following we assume that the calculus is equipped with the type system defined by [18], which is variant of the Hindley-Milner type system [16] 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 $\texttt{field}(L)$ to a neighbouring field of elements of type L, and correspondingly a type T to any expression.

Remark 1. *Figure 2 presents the collection of built-in functions and operators used in this paper (a small subset of possible built-in functions covered by this calculus). A few notes regarding these functions:*

- *Recall that each built-in function with local arguments is overloaded to work on fields on a pointwise basis.*

- *The multiplex operator* $\texttt{mux}$ *selects between its second and third arguments based on the value of the first one. This is similar to the* $\texttt{if}$ *keyword but not equivalent:* $\texttt{mux}$ *evaluates both of these arguments everywhere, whereas* $\texttt{if}$ *only evaluates each on the subspace with the matching Boolean value.*

- *A special role is played by the second-order operator* ***foldHood*** *and its specialisations for different aggregation functions (**minHood**, **maxHood** and so on) that collapse a field value into a local value (reminiscent of "reduce" functions common in parallel programming frameworks like MPI). The versions of these operators ending in* + *also aggregate the value corresponding to the current device (which is otherwise ignored), while the versions ending in* ***Loc*** *also aggregate a given local value in place of the value corresponding to the current device.*

Example 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, computed as length of a minimum path that circumvents an `obstacle` area:

```
def distanceToWithObs(source, obstacle) {
  if(obstacle) { infinity } { distanceTo(source) }
}

def distanceTo(source) {
  mux( source, 0,
    rep (infinity) { (x) => minHood(nbr{x} + nbrRange())}
  )
}
```

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`, via a purely functional `mux` built-in operator, we give 0 on sources (i.e., on devices evaluating `source` to `True`). On other devices, we compute the estimated as distance being `infinity` at the beginning, then evolving the distance esimate 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 built-in `nbrRange` modeling a local range sensor).

3.2   Operational Semantics

We now present a formal semantics that can serve both as a specification for implementation of programming languages based on the calculus and for reasoning about its properties. Differently from models like BSP [48] that can enact system-wide synchronous rounds in which each device computes exactly once, in our model individual devices undergo computation in (local) rounds, which are sequential for each device, and interleaved among different devices. 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. The scheduling of such rounds across the network is fair and asynchronous—the considered notion of fairness is explained in Section 4.1, and basically amounts to the eventual existence of another round for each device and for each moment of time. To simplify the notation, we shall assume a fixed program P. We say that "device δ *fires*", to mean that the main expression e_{main} of P is evaluated on δ at a particular round.

Network evolution is modelled (in Section 3.2.2) by a small-step semantics, given as a transition system $\xrightarrow{act}$ on network configurations N, where actions can either be firings of a device or network configuration changes. The semantics of a firing action is defined in terms of the computation that takes place on an individual device, which is modelled (in Section 3.2.1) by a big-step semantics. Note that we use small-step semantics in network transitions to capture the step-by-step evolution of a network, while the more abstract big-step semantics

is used in individual devices since in that case only the final result of round computation matters—and is in fact unique.

3.2.1 Device Semantics

The computation that takes place on a single device is formalised by a big-step semantics, expressed by the judgement $\delta; \Theta \vdash \mathtt{e_{main}} \Downarrow \theta$, to be read "expression $\mathtt{e_{main}}$ evaluates to θ on device δ with respect to environment Θ". The result of evaluation is a *value-tree* θ, which is an ordered tree of values that tracks the results of all evaluated subexpressions of $\mathtt{e_{main}}$. Such a result is made available to δ's neighbours for their subsequent firing (including δ itself, so as to support a form of state across computation rounds). The recently-received value-trees of neighbours are then collected into a *value-tree environment* Θ, implemented as a map from device identifiers to value-trees (written $\bar{\delta} \mapsto \bar{\theta}$ as short for $\delta_1 \mapsto \theta_1, \ldots, \delta_n \mapsto \theta_n$). Intuitively, the outcome of the evaluation will depend on those value-trees. Figure 3 (top) defines value-trees and value-tree environments—the syntax of values $\mathtt{v}$ is given in Fig. 1.

Example 2. *The graphical representation of the value trees* $5\langle 2\langle\rangle, 3\langle\rangle\rangle$ *and* $5\langle 2\langle\rangle, 3\langle 7\langle\rangle, 1\langle\rangle, 4\langle\rangle\rangle\rangle$ *is as follows:*

```
    5                   5
   / \                 / \
  2   3               2   3
                         /|\
                        7 1 4
```

In the following, for sake of readability, we sometimes write the value $\mathtt{v}$ as short for the value-tree $\mathtt{v}\langle\rangle$. Following this convention, the value-tree $5\langle 2\langle\rangle, 3\langle\rangle\rangle$ is shortened to $5\langle 2, 3\rangle$, and the value-tree $5\langle 2\langle\rangle, 3\langle 7\langle\rangle, 4\langle\rangle, 4\langle\rangle\rangle\rangle$ is shortened to $5\langle 2, 3\langle 7, 1, 4\rangle\rangle$.

Figure 3 (bottom) defines the judgement $\delta; \Theta \vdash \mathtt{e} \Downarrow \theta$, where: *(i)* δ is the identifier of the current device; *(ii)* Θ is the neighbouring field of the value-trees produced by the most recent evaluation of (an expression corresponding to) $\mathtt{e}$ on δ's neighbours; *(iii)* $\mathtt{e}$ is a closed run-time expression (i.e., a closed expression that may contain neighbouring field values); *(iv)* the value-tree θ represents the values computed for all the expressions encountered during the evaluation of $\mathtt{e}$—in particular the root of the value tree θ, denoted by $\rho(\theta)$, is the value computed for expression $\mathtt{e}$. The auxiliary function ρ is defined in Figure 3 (second frame).

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression $\mathtt{e}'$ of $\mathtt{e}$ with respect to the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ. This process, called *alignment*, is modelled by the auxiliary function π defined in Figure 3 (second frame). This function has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i-th subtree of θ; while $\pi^\ell(\theta)$ extracts the last subtree of θ, *if* the root of the first subtree of θ is equal to the local (boolean) value ℓ (thus implementing a filter specifically designed for the $\mathtt{if}$ construct). Auxiliary functions ρ and π apply pointwise on value-tree environments, as defined in Figure 3 (second frame).

Rules [E-LOC] and [E-FLD] model the evaluation of expressions that are either a local value or a neighbouring field value, respectively: note that in [E-FLD] we

Value-trees and value-tree environments:

$$\theta \quad ::= \quad \mathtt{v}\langle\overline{\theta}\rangle \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{value-tree}$$
$$\Theta \quad ::= \quad \overline{\delta} \mapsto \overline{\theta} \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{value-tree environment}$$

Auxiliary functions:

$$\rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$$
$$\pi_i(\mathtt{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_i \quad \text{if } 1 \leq i \leq n \qquad \pi^\ell(\mathtt{v}\langle\theta_1,\theta_2\rangle) = \theta_2 \quad \text{if } \rho(\theta_1) = \ell$$
$$\pi_i(\theta) = \bullet \quad \text{otherwise} \qquad\qquad\qquad \pi^\ell(\theta) = \bullet \quad \text{otherwise}$$

$$\text{For } aux \in \rho, \pi_i, \pi^\ell : \begin{cases} aux(\delta \mapsto \theta) &= \delta \mapsto aux(\theta) & \text{if } aux(\theta) \neq \bullet \\ aux(\delta \mapsto \theta) &= \bullet & \text{if } aux(\theta) = \bullet \\ aux(\Theta, \Theta') &= aux(\Theta), aux(\Theta') \end{cases}$$

$$args(\mathtt{d}) = \overline{\mathtt{x}} \quad \text{if } \mathtt{def}\ \mathtt{d}(\overline{\mathtt{x}})\ \{\mathtt{e}\} \qquad body(\mathtt{d}) = \mathtt{e} \quad \text{if } \mathtt{def}\ \mathtt{d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$$

Syntactic shorthands:

$$\delta; \overline{\pi}(\Theta) \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \text{ where } |\overline{\mathtt{e}}| = n \text{ for } \delta; \pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \theta_1 \cdots \delta; \pi_n(\Theta) \vdash \mathtt{e}_n \Downarrow \theta_n$$
$$\rho(\overline{\theta}) \qquad\qquad \text{where } |\overline{\theta}| = n \text{ for } \rho(\theta_1), \ldots, \rho(\theta_n)$$
$$\overline{\mathtt{x}} := \rho(\overline{\theta}) \qquad \text{where } |\overline{\mathtt{x}}| = n \text{ for } \mathtt{x}_1 := \rho(\theta_1) \ldots \mathtt{x}_n := \rho(\theta_n)$$

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

$$[\text{E-LOC}] \frac{}{\delta; \Theta \vdash \ell \Downarrow \ell\langle\rangle} \qquad\qquad [\text{E-FLD}] \frac{\phi' = \phi|_{\mathbf{dom}(\Theta) \cup \{\delta\}}}{\delta; \Theta \vdash \phi \Downarrow \phi'\langle\rangle}$$

$$[\text{E-LET}] \frac{\delta; \pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \theta_1 \qquad \delta; \pi_2(\Theta) \vdash \mathtt{e}_2[\mathtt{x} := \rho(\theta_1)] \Downarrow \theta_2}{\delta; \Theta \vdash \mathtt{let}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}$$

$$[\text{E-B-APP}] \frac{\delta; \overline{\pi}(\Theta) \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \qquad \mathtt{v} = (\!|\mathtt{b}|\!)_\delta^\Theta (\rho(\overline{\theta}))}{\delta; \Theta \vdash \mathtt{b}(\overline{\mathtt{e}}) \Downarrow \mathtt{v}\langle\overline{\theta}\rangle}$$

$$[\text{E-D-APP}] \frac{\delta; \overline{\pi}(\Theta) \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \qquad \delta; \Theta \vdash body(\mathtt{d})[args(\mathtt{d}) := \rho(\overline{\theta})] \Downarrow \theta'}{\delta; \Theta \vdash \mathtt{d}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\overline{\theta}, \theta'\rangle}$$

$$[\text{E-NBR}] \frac{\delta; \pi_1(\Theta) \vdash \mathtt{e} \Downarrow \theta \qquad \phi = \rho(\pi_1(\Theta))[\delta \mapsto \rho(\theta)]}{\delta; \Theta \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \phi\langle\theta\rangle}$$

$$[\text{E-REP}] \frac{\begin{array}{c} \delta; \pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \theta_1 \\ \delta; \pi_2(\Theta) \vdash \mathtt{e}_2[\mathtt{x} := \ell_0] \Downarrow \theta_2 \end{array} \quad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \rho(\theta_1) & \text{otherwise} \end{cases}}{\delta; \Theta \vdash \mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x}) \texttt{=>} \mathtt{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}$$

$$[\text{E-IF}] \frac{\delta; \pi_1(\Theta) \vdash \mathtt{e} \Downarrow \theta_1 \quad \rho(\theta_1) \in \{\mathtt{True}, \mathtt{False}\} \quad \delta; \pi^{\rho(\theta_1)}(\Theta) \vdash \mathtt{e}_{\rho(\theta_1)} \Downarrow \theta}{\delta; \Theta \vdash \mathtt{if}(\mathtt{e})\{\mathtt{e}_{\mathtt{True}}\}\{\mathtt{e}_{\mathtt{False}}\} \Downarrow \rho(\theta)\langle\theta_1, \theta\rangle}$$

Figure 3: Big-step operational semantics for expression evaluation.

take care of restructing the domain of a neighbouring field value to the only set of neighbour devices as reported in Θ.

Rule [E-LET] is fairly standard: it first evaluates $\mathtt{e}_1$ and then evaluates the expression obtained from $\mathtt{e}_2$ by replacing all the occurrences of the variable $\mathtt{x}$ with the value of $\mathtt{e}_1$.

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $\mathtt{b}(\mathtt{e}_1 \cdots \mathtt{e}_n)$, where $n \geq 0$. It produces the value-tree $\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle$, where $\theta_1, \ldots, \theta_n$ are the value-trees produced by the evaluation of the actual parameters $\mathtt{e}_1, \ldots, \mathtt{e}_n$ and $\mathtt{v}$ is the value returned by

the function. The rule exploits the special auxiliary function $(\!|b|\!)_\delta^\Theta$, whose actual definition is abstracted away. This is such that $(\!|b|\!)_\delta^\Theta(\overline{v})$ computes the result of applying built-in function b to values $\overline{v}$ in the current environment of the device δ. In particular: the built-in 0-ary function $\texttt{uid}$ gets evaluated to the current device identifier (i.e., $(\!|\texttt{uid}|\!)_\delta^\Theta() = \delta$), and mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $(\!|+|\!)_\delta^\Theta(2,3) = 5$).

Example 3. *Evaluating the expression $+(2,3)$ produces the value-tree $5\langle 2,3\rangle$. The value of the whole expression, 5, has been computed by using rule [E-B-APP] to evaluate the application of the sum operator $+$ to the values 2 (the root of the first subtree of the value-tree) and 3 (the root of the second subtree of the value-tree).*

The $(\!|b|\!)_\delta^\Theta$ function also encapsulates measurement variables such as $\texttt{nbrRange}$ and interactions with the external world via sensors and actuators.

Rule [E-D-APP] models the application of a user-defined function. It is used to evaluate expressions of the form $d(e_1 \ldots e_n)$, where $n \geq 0$. It resembles rule [E-B-APP] while producing a value-tree with one more subtree θ', which is produced by evaluating the body of the function d (denoted by $body(d)$) substituting the formal parameters of the function (denoted by $args(d)$) with the values obtained evaluating $e_1, \ldots e_n$.

Rule [E-REP] implements internal state evolution through computational rounds: $\texttt{rep}(e_1)\{(x)\texttt{=>}e_2\}$ evaluates to $e_2[x := v]$ where v is obtained from e_1 on the first firing of a device, from the previous value of the whole $\texttt{rep}$-expression otherwise.

Example 4. *To illustrate rule [E-REP], as well as computational rounds, we consider program $\texttt{rep(0)\{ (x) => +(x, 1)\}}$. The first firing of a device δ is performed against the empty tree environment. Therefore, according to rule [E-REP], to evaluate $\texttt{rep(0)\{ (x) => +(x, 1)\}}$ means to evaluate the subexpression $\texttt{+(0, 1)}$, obtained from $\texttt{+(x, 1)}$ by replacing x with 0. This produces the value-tree $\theta = 1\langle 0, 1\langle 0, 1\rangle\rangle$, where root 1 is the overall result as usual, while its subtrees are the result of evaluating the first and second argument respectively. Any subsequent firing of the device δ is performed with respect to a tree environment Θ that associates to δ the outcome θ of the most recent firing of δ. Therefore, evaluating $\texttt{rep(0)\{ (x) => +(x, 1)\}}$ at the second firing means to evaluate the subexpression $\texttt{+(1, 1)}$, obtained from $\texttt{+(x, 1)}$ by replacing x with 1, which is the root of θ. Hence the results of computation are 1, 2, 3, and so on.*

Rule [E-NBR] models device interaction. It first collects neighbour's values for expressions e as $\phi = \rho(\pi_1(\Theta))$, then evaluates e in δ and updates the corresponding entry in ϕ.

Example 5. *To illustrate rule [E-NBR], consider the program:*

$$e' = \texttt{minHood(nbr\{snsNum()\})}$$

where the 1-ary built-in function $\texttt{minHood}$ returns the lower limit of values in the range of its neighbouring field argument, and the 0-ary built-in function $\texttt{snsNum}$ returns the numeric value measured by a sensor. Suppose that the program runs on a network of three devices δ_A, δ_B, and δ_C where:

- *δ_B and δ_A are mutually connected, δ_B and δ_C are mutualy connected, while δ_A and δ_C are not connected;*

- `snsNum` *returns 1 on δ_A, 2 on δ_B, and 3 on δ_C; and*

- *all devices have an initial empty tree-environment $\emptyset$.*

Suppose that device δ_A is the first device that fires: the evaluation of `snsNum()` on δ_A yields 1 (by rules [E-LOC] and [E-B-APP], since $(\!|\texttt{snsNum}|\!)^{\emptyset}_{\delta_A}() = 1$); the evaluation of `nbr{snsNum()}` on δ_A yields $(\delta_A \mapsto 1)\langle 2 \rangle$ (by rule [E-NBR]); and the evaluation of e' on δ_A yields

$$\theta_A \quad = \quad 1\langle(\delta_A \mapsto 1)\langle 1 \rangle\rangle$$

(by rule [E-B-APP], since $(\!|\texttt{minHood}|\!)^{\emptyset}_{\delta_A}(\delta_A \mapsto 1) = 1$). Therefore, at its first fire, device δ_A produces the value-tree θ_A. Similarly, if device δ_C is the second device that fires, it produces the value-tree

$$\theta_C \quad = \quad 3\langle(\delta_C \mapsto 3)\langle 3 \rangle\rangle$$

Suppose that device δ_B is the third device that fires. Then the evaluation of e' on δ_B is performed with respect to the value tree environment $\Theta_B = (\delta_A \mapsto \theta_A,\ \delta_C \mapsto \theta_C)$ and the evaluation of its subexpressions `nbr{snsNum()}` and `snsNum()` is performed, respectively, with respect to the following value-tree environments obtained from Θ_B by alignment:

$$\begin{aligned}
\Theta'_B &= \pi_1(\Theta_B) = (\delta_A \mapsto (\delta_A \mapsto 1)\langle 1 \rangle,\ \ \delta_C \mapsto (\delta_C \mapsto 3)\langle 3 \rangle) \\
\Theta''_B &= \pi_1(\Theta'_B) = (\delta_A \mapsto 1,\ \ \delta_C \mapsto 3)
\end{aligned}$$

We thus have that $(\!|\texttt{snsNum}|\!)^{\Theta''_B}_{\delta_B}() = 2$; the evaluation of `nbr{snsNum()}` on δ_B with respect to Θ'_B produces the value-tree $\phi\langle 2 \rangle$ where $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$; and $(\!|\texttt{minHood}|\!)^{\Theta_B}_{\delta_B}(\phi) = 1$. Therefore the evaluation of e' on δ_B produces the value-tree $\theta_B = 1\langle\phi\langle 2 \rangle\rangle$. Note that, if the network topology and the values of the sensors will not change, then: any subsequent fire of device δ_B will yield a value-tree with root 1 (which is the minimum of `snsNum` across δ_A, δ_B and δ_C); any subsequent fire of device δ_A will yield a value-tree with root 1 (which is the minimum of `snsNum` across δ_A and δ_B); and any subsequent fire of device δ_C will yield a value-tree with root 2 (which is the minimum of `snsNum` across δ_B and δ_C).

Rule [E-IF] is almost standard, except that it performs domain restriction $\pi^{\texttt{True}}(\Theta)$ (resp. $\pi^{\texttt{False}}(\Theta)$) in order to guarantee that subexpression $e_{\texttt{True}}$ is not matched against value-trees obtained from $e_{\texttt{False}}$ (and vice-versa).

3.2.2 Network Semantics

The overall network evolution is formalised by the small-step operational semantics given in Figure 4 as a transition system on network configurations N. Figure 4 (top) defines key syntactic elements to this end. Ψ models the overall status of the devices in the network at a given time, as a map from device identifiers to value-tree environments. From it, we can define the state of the field at that time by summarising the current values held by devices. τ models *network topology*, namely, a directed neighbouring graph, as a map from device identifiers to set of identifiers (denoted as I). Σ models *sensor (distributed) state*, as a map from device identifiers to (local) sensors (i.e., sensor name/value

System configurations and action labels:

$$
\begin{array}{lll}
\Psi & ::= & \overline{\delta} \mapsto \overline{\Theta} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{status field} \\
\tau & ::= & \overline{\delta} \mapsto \overline{I} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\text{topology} \\
\Sigma & ::= & \overline{\delta} \mapsto \overline{\sigma} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{sensors-map} \\
Env & ::= & \tau, \Sigma \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{environment} \\
N & ::= & \langle Env; \Psi \rangle \qquad\qquad\qquad\qquad\qquad\qquad\text{network configuration} \\
act & ::= & \delta \mid env \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\text{action label}
\end{array}
$$

Environment well-formedness:

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

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

$$
\text{[N-FIR]} \quad \frac{Env = \tau, \Sigma \qquad \delta; F(\Psi)(\delta) \vdash \mathtt{e_{main}} \Downarrow \theta \ (\text{w.r.t. } \Sigma(\delta)) \qquad \Psi_1 = \overline{\delta} \mapsto \{\delta \mapsto \theta\}}{\langle Env; \Psi \rangle \xrightarrow{\delta} \langle Env; F(\Psi)[\Psi_1] \rangle}
$$
$$
\tau(\delta) = \overline{\delta}
$$

$$
\text{[N-ENV]} \quad \frac{WFE(Env') \qquad Env' = \tau, \overline{\delta} \mapsto \overline{\sigma} \qquad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle}
$$

Figure 4: Small-step operational semantics for network evolution.

maps denoted as σ). Then, Env (a couple of topology and sensor state) models the system's environment. So, a whole network configuration N is a couple of a status field and environment.

We use the following notation for status fields. Let $\overline{\delta} \mapsto \Theta$ denote a map from device identifiers $\overline{\delta}$ to the same value-tree environment Θ. Let $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\mathbf{dom}(\Theta_0) \cup \mathbf{dom}(\Theta_1)$ coinciding with Θ_1 in the domain of Θ_1 and with Θ_0 otherwise. Let $\Psi_0[\Psi_1]$ denote the status field with the *same domain* as Ψ_0 made of $\delta \mapsto \Psi_0(\delta)[\Psi_1(\delta)]$ for all δ in the domain of Ψ_1, $\delta \mapsto \Psi_0(\delta)$ otherwise.

We consider transitions $N \xrightarrow{act} N'$ of two kinds: firings, where act is the corresponding device identifier, and environment changes, where act is the special label env. This is formalised in Figure 4 (bottom). Rule [N-FIR] models a computation round (firing) at device δ: it takes the local value-tree environment filtered out of old values $F(\Psi)(\delta)$;[3] then by the single device semantics it obtains the device's value-tree θ,[4] which is used to update the system configuration of δ and of δ's neighbours.

Rule [N-ENV] takes into account the change of the environment to a new *well-formed* environment Env'—environment well-formedness is specified by the predicate $WFE(Env)$ in Figure 4 (middle). Let $\overline{\delta}$ be the domain of Env'. We first construct a status field Ψ_0 associating to all the devices of Env' the empty context $\emptyset$. Then, we adapt the existing status field Ψ to the new set of devices:

---

[3]Function $F(\Psi)$ in rule [N-FIR] models a filtering operation that clears out old stored values from the value-tree environments in Ψ, implicitly based on space/time tags.

[4]We shall assume that any device firing is guaranteed to terminate in any environmental condition. Termination of a device firing is clearly not decidable, but we shall assume—without loss of generality for the results of this paper—that a decidable subset of the termination fragment can be identified (e.g., by ruling out recursive user-defined functions or by applying standard static analysis techniques for termination).

16

$\Psi_0[\Psi]$ automatically handles removal of devices, map of new devices to the empty context, and retention of existing contexts in the other devices.

Example 6. *Consider a network of devices with the program*

$$\mathtt{e}' = \mathtt{minHood(nbr\{snsNum()\})}$$

introduced in Example 5. The network configuration illustrated at the beginning of Example 5 can be generated by applying rule [N-ENV] to the empty network configuration. I.e., we have

$$\langle \emptyset, \emptyset; \emptyset \rangle \xrightarrow{env} \langle Env_0; \Psi_0 \rangle$$

where

- $Env_0 = \tau_0, \Sigma_0$,

- $\tau_0 = (\delta_A \mapsto \{\delta_B\}, \delta_B \mapsto \{\delta_A, \delta_C\}, \delta_C \mapsto \{\delta_B\})$,

- $\Sigma_0 = (\delta_A \mapsto (\mathtt{snsNum} \mapsto 1), \delta_B \mapsto (\mathtt{snsNum} \mapsto 2), \delta_C \mapsto (\mathtt{snsNum} \mapsto 3))$, *and*

- $\Psi_0 = (\delta_A \mapsto \emptyset, \delta_B \mapsto \emptyset, \delta_C \mapsto \emptyset)$.

Then, the tree fires of devices δ_A, δ_C and δ_B illustrated in Example 5 correspond to the following transitions, respectively.

1. $\langle Env_0; \Psi_0 \rangle \xrightarrow{\delta_A} \langle Env_0; \Psi' \rangle$, *where*

    - $\Psi' = (\delta_A \mapsto (\delta_A \mapsto \theta_A),\ \delta_B \mapsto (\delta_A \mapsto \theta_A),\ \delta_C \mapsto \emptyset)$, *and*

    - $\theta_A = 1\langle(\delta_A \mapsto 1)\langle 1\rangle\rangle$;

2. $\langle Env_0; \Psi' \rangle \xrightarrow{\delta_C} \langle Env_0; \Psi'' \rangle$, *where*

    - $\Psi'' = (\delta_A \mapsto (\delta_A \mapsto \theta_A),\ \delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_C \mapsto \theta_C),\ \delta_C \mapsto (\delta_C \mapsto \theta_C))$, *and*

    - $\theta_C = 1\langle(\delta_C \mapsto 3)\langle 3\rangle\rangle$;

3. $\langle Env_0; \Psi'' \rangle \xrightarrow{\delta_B} \langle Env_0; \Psi''' \rangle$, *where*

    - $\Psi''' = (\delta_A \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B),$
      $\qquad \delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C),$
      $\qquad \delta_C \mapsto (\delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C))$,

    - $\theta_B = 1\langle\phi\langle 2\rangle\rangle$, *and*

    - $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$.

3.3 A Minimal Convenient Extension: Functional Parametrisation

The pragmatic convenience of the calculus defined so far can be improved to express general-purpose *building blocks*, which are parametric algorithms designed to be applied to a broad class of problems, and necessarily make use of functional parameters to tune their behaviour.

To this end, we extend the syntax of user-defined functions to admit *functional parameters*, ranged over by z. Such *extended functions* can be defined as def d($\bar{\mathrm{x}}$)($\bar{\mathrm{z}}$){e} and called by d($\bar{\mathrm{e}}$)($\bar{\mathrm{f}}$) where the arguments $\bar{\mathrm{f}}$ 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. By convention, we omit the second parentheses whenever no functional parameters are present; so that functions without functional parameters can be defined and called as usual. We also allow the presence of built-in functions admitting functional parameters (e.g., the field aggregator foldhood(x, y)(z) which combines values in a field x through an initial value y and a binary function z given as functional parameter).

We remark that a functional parameter z (like any other function name) is not an expression by itself, and it only constitutes one when provided with appropriate arguments or passed as argument to a function. This implies for instance that (if(e){z_1}{z_2})($\bar{\mathrm{e}}$) is *not* a valid expression.

A program in the extended syntax can be converted to a program in plain first-order syntax by systematically substituting each call d($\bar{\mathrm{e}}$)($\bar{\mathrm{f}}$) to an extended function def d($\bar{\mathrm{x}}$)($\bar{\mathrm{z}}$){e} (where the arguments $\bar{\mathrm{f}}$ do not contain functional parameters) by a call $\mathrm{d}_{\bar{\mathrm{f}}}$($\bar{\mathrm{e}}$) to a plain function $\mathrm{d}_{\bar{\mathrm{f}}}$ defined as def $\mathrm{d}_{\bar{\mathrm{f}}}$($\bar{\mathrm{x}}$){e[$\bar{\mathrm{z}}$:= $\bar{\mathrm{f}}$]}—thus interpreting functional parameters as macro parameters.[5] For example, the following program (comparing minimum temperature and maximum threshold across a network):

```
def foldwithlocal(field, local, initial)(aggregate) {
  aggregate(foldHood(field, initial)(aggregate), local)
}

def gossip(null)(aggregate, sensor) {
  rep (initial) { (x) => foldwithlocal(nbr{x}, sensor(), initial)(aggregate) }
}

gossip(infinity)(min, sns_temp) < gossip(-infinity)(max, sns_threshold)
```

can be rewritten eliminating functional parameters in the following way:

```
def foldwithlocal_min(field, local, initial) {
  min(foldHood_min(field, initial), local)
}

def gossip_min_temp(initial) {
  rep (initial) { (x) => foldwithlocal_min(nbr{x}, sns_temp(), initial) }
}

def foldwithlocal_max(field, local, initial) {
  max(foldHood_max(field, initial), local)
}

def gossip_max_threshold(initial) {
  rep (initial) { (x) => foldwithlocal_max(nbr{x}, sns_threshold(),initial) }
}

gossip_min_temp(infinity) < gossip_max_threshold(-infinity)
```

where foldHood_min and foldHood_max can then be substituted with their equivalent versions minHood, maxHood.

---

[5]This rewriting process always terminates. Consider F as the set of distinct plain function names that are passed as parameters to extended functions in any point of the program. Then an extended function with k functional parameters can be instantiated at most once for each combination of functions in F, that is, at most n^k times where n is the cardinality of F.

3.4 Examples: Building Blocks

Through functional parametrisation we are now able to express the main "building blocks" used in field calculus (as reported in [9]), 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:

3.4.1 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: first, computing a field of shortest-path distances from the source region according to the supplied function `metric`, and second, 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 $\texttt{minHoodLoc}(\phi, \ell)$, which selects the minimum of the neighbours' values in ϕ and the local value ℓ (i.e. the minimum in $\phi[\delta \mapsto \ell]$) 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), 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)) }
```

3.4.2 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)) }
```

3.4.3 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 the 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 [47]), we use the definition from [24] as further restricted by [17]: 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) focuses 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 [38] and computations that converge only in the limit like Laplacian-based approximate consensus [42]. 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 [21]). Preliminary investigations in this area have produced positive results (e.g., [21, 39]), 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 defined 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:*

- stabilising *iff every fair sequence stabilises given fixed environmental conditions Env;*

- self-stabilising *to state N iff every fair sequence stabilises to the same state N given fixed environmental conditions Env.*

A function $\mathtt{f}(\overline{\mathtt{x}})$ *is* self-stabilising *iff given any self-stabilising expressions* $\overline{\mathtt{e}}$ *of the type of the inputs of* f *the expression* $\mathtt{f}(\overline{\mathtt{e}})$ *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 reach a stable state only when environmental changes are transitory.

4.2 Eventual Behaviour

Define a *computational field* Φ as a map $\overline{\delta} \mapsto \overline{\mathtt{v}}$,[6] such that if $\overline{\mathtt{v}}$ have field type their domains are *coherent* with the environment $\langle \tau, \Sigma \rangle$, 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 $\mathcal{T}[\![T]\!] = \mathbf{D} \rightharpoonup \mathcal{V}[\![T]\!]$ be[7] 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 by cases, and executed in the restricted environment corresponding to $\mathbf{dom}(\Phi)$)—in which case we say that e is a *self-stabilising expression for* Φ.

Note that a network status Ψ induces uniquely a computational field Φ defined by $\Phi(\delta) = \rho(\Psi(\delta)(\delta))$, while conversely each Φ is coherent with multiple network statuses Ψ. Thus a computational field is not sufficient to capture the

---

[6]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.

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

whole status of a computation of a program P. However, for self-stabilising programs P and self-stabilising functions 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 $f(\overline{e})$ where $\overline{e}$ are self-stabilising expressions for $\overline{\Phi}$. Since 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[8].

Definition 2 (Eventual behaviour). *Let* e *be a self-stabilising closed expression. We write* $[\![e]\!]$ *for the computational field* Φ *eventually produced by the computation of* e.

Let f *be a self-stabilising function of type* $\overline{T} \to T'$, *where* $\overline{T} = T_1 \times \cdots \times T_n$ ($n \geq 0$). *We write* $[\![f]\!]$ *for the mathematical function in* $(\mathcal{T}[\![T_1]\!] \times \cdots \times \mathcal{T}[\![T_n]\!]) \to \mathcal{T}[\![T']\!]$,[9] *such that* $[\![f]\!](\overline{\Phi}) = [\![f(\overline{e})]\!]$ *where* $\overline{e}$ *are self-stabilising expressions for* $\overline{\Phi}$.

Using the above definition of eventual behaviour, it is possible to precisely state and investigate the equivalence of different self-stabilising programs.

Proposition 1 (Eventual behaviour preserving equivalences). *1. Let* e_1, e_2 *be self-stabilising expressions with the same eventual behaviour. Then given a self-stabilising expression* e, *swapping* e_1 *for* e_2 *in* e *does not change the eventual outcome of its computation.*

2. Let f_1, f_2 *be self-stabilising functions with the same eventual behaviour. Then given a self-stabilising expression* e, *swapping* f_1 *for* f_2 *in* e *does not change the eventual outcome of its computation.*

3. Let e *be a self-stabilising expression calling a user-defined self-stabilising function* d *such that in* body(f) *no* $x \in args(f)$ *occurs in the branch of an* if. *Let* e′ *be the expression obtained from* e *by substituting each function application of the kind* $f(\overline{e})$ *with* body(f)$[args(f) := \overline{e}]$. *Then* e′ *is self-stabilising and has the same eventual behaviour as* e *(i.e.* $[\![e]\!] = [\![e′]\!]$).

Proof. 1. By straightforward induction on the structure of an expression. The base case is given by expressions without occurrences of e_1 and e_2, and by expressions e_i for $i = 1, 2$. The inductive step follows by compositionality of the operational semantics.

2. For the same reasoning as in point (1), where the base case is given by expressions without occurrences of f_1 and f_2 and by expressions $f_i(\overline{e})$ for $i = 1, 2$.

3. Recall that no expressions with side effects are contemplated in the present calculus. Since no argument of f occurs in the branch of an if, each of those arguments is evaluated in the same environment as the whole function application $f(\overline{e})$. It follows that $e_1 = f(\overline{e})$ and $e_2 = body(f)[args(f) :=$

---

[8]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 [48].

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

$\overline{e}$] have the same behaviour (hence the same eventual behaviour). The thesis follows then by applying point (1) to expressions e_1 and e_2.

$\square$

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:

1. identifying a "base" fragment of the Field Calculus that contains only self-stabilising programs;

2. identifying a set of eventual behaviour preserving equivalences (cf. Proposition 1);

3. relying on the fact that the least fragment of the Field Calculus that contains the fragment (1) and is closed under the equivalences (2) 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 with equivalence results further extending the fragment (in Section 5.3), and finally discuss its expressiveness (in Section 5.4).

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.

Example 7. *Consider the following function:*

```
def f1(v) { rep (v) { (x) => v-x } }
```

This function 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, . . . *Thus in this case self-stabilisation is prevented by an* oscillating *behaviour.*

Example 8. *Consider the following function (a classical gossip implementation):*

```
def f2(v) { rep (v) { (x) => max(maxHood+(nbr{x}), v) } }
```

This function 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*

$$s \quad ::= \quad \texttt{x} \mid \texttt{v} \mid \texttt{let x} = \texttt{s in s} \mid \texttt{f}(\overline{\texttt{s}}) \mid \texttt{if(s)\{s\}\{s\}} \mid \texttt{nbr\{s\}} \qquad \text{self-stabilising}$$
$$\mid \texttt{rep(e)\{(x)=>f}^{\texttt{C}}\texttt{(nbr\{x\},nbr\{s\},}\overline{\texttt{e}}\texttt{)\}} \qquad \text{expression}$$
$$\mid \texttt{rep(e)\{(x)=>f(mux(nbrlt(s),nbr\{x\},s),}\overline{\texttt{s}}\texttt{)\}}$$
$$\mid \texttt{rep(e)\{(x)=>f}^{\texttt{R}}\texttt{(minHoodLoc(f}^{\texttt{MP}}\texttt{(nbr\{x\},}\overline{\texttt{s}}\texttt{),s),x,}\overline{\texttt{e}}\texttt{)\}}$$

Figure 5: 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.

value k was given in some device, the eventual output of the function upon a fixed input $v < k$ 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".

Example 9. *Consider the following function, with input v of an unbounded integer type (big integer):*

```
def f3(v) { rep (v) { (x) => min(minHood(nbr{x}) - 1, v) } }
```

This function 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.*

Oscillation, state preservation, and divergence are three key causes of non self-stabilisation that the proposed fragment will address.

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 5. This fragment includes:

- all expressions not containing a `rep` construct, hence comprising built-in functions, which are therefore assumed to be self-stabilising;

- three special forms of `rep`-constructs, defined with a specific syntax coupled with semantic restrictions on relevant functional parameters.

5.2.1 The $\textsf{C}, \textsf{M}, \textsf{P}, \textsf{R}$ 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 $\textsf{M}$, $\textsf{P}$, and $\textsf{R}$ require some of their argument types to be equipped with a *partial order* relation, while property $\textsf{C}$ 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 $f(\phi, \psi, \overline{v})$ 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 $\phi, \psi, \overline{v}$ coherent with the domain of Θ, and assuming that $\delta; \Theta \vdash f(\phi, \psi, \overline{v}) \Downarrow \ell\langle\overline{\theta}\rangle$:

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

where dist is any metric.

Example 10. *Function* $f_1(\phi, \psi) = pickHood(\psi - \phi) = (\psi - \phi)(\delta)$ *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* $f_2(\phi, \psi) = pickHood((\psi + \phi)/2)$ *and* $f_3(\phi, \psi) = pickHood(\psi) + meanHood(\phi - \psi)/2$ *are converging.*

M **(Monotonic non-decreasing)** A stateless[10] function $f(x, \overline{x})$ with arguments of local type is monotonic non-decreasing in its first argument iff whenever $\ell_1 \leq \ell_2$, also $f(\ell_1, \overline{\ell}) \leq f(\ell_2, \overline{\ell})$.

Example 11. *Function* $f_1(\ell) = \ell - 1$ *is monotonic non-decreasing, while function* $f_2(\ell) = \ell^2$ *is not.*

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

Example 12. *Function* $f_1(\ell) = \ell + 1$ *is progressive, while functions* $f_2(\ell) = \ell - 1$, $f_3(\ell) = \ell^2$ *are not.*

R **(Raising)** A function $f(\ell_1, \ell_2, \overline{v})$ is raising with respect to partial orders $<$, $\lhd$ iff:

- $f(\ell, \ell, \overline{v}) = \ell$;

- $f(\ell_1, \ell_2, \overline{v}) \geq \min(\ell_1, \ell_2)$;

- either $f(\ell_1, \ell_2, \overline{v}) \rhd \ell_2$ or $f(\ell_1, \ell_2, \overline{v}) = \ell_1$.

Example 13. *Function* $f_1(\ell_1, \ell_2) = \ell_1$ *is raising with respect to any partial orders. Function* $f_2(\ell_1, \ell_2) = \ell_1 - \ell_2$ *is not raising since it violates both the first two clauses. Function* $f_3(\ell_1, \ell_2) = (\ell_1 + \ell_2)/2$ *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.

---

[10]A function $f(\overline{x})$ is *stateless* iff given fixed inputs $\overline{v}$ 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.

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 $\mathtt{\overline{e}}$. If the range of the metric granting convergence is a well-founded set[11] of real numbers, the pattern self-stabilises since it gradually approaches the value given by `s`.

Example 14. *Function* `f1` *in Example 7 does not respect the converging* `rep` *pattern, as shown in Example 10. However, if we change* `f1` *to the following and assume that its input and output are finite-precision numeric values (e.g., Java's* `double`*):*

```
def filter(v) { rep (v) { (x) => (v+x)/2 } }
```

we obtain a low-pass filter *that is self-stabilising and complies with the converging* `rep` *pattern.*

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 15. *Function* `f2` *in Example 8 does not respect the acyclic* `rep` *pattern, since it aggregates all neighbours without any "acyclic filtering". However, if we change* `f2` *to the following:*

```
def f2C(v, p) { rep (v) { (x) => max(maxHood+(mux(nbrlt(p), nbr{x}, 0), v) } }
```

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*.[12]

---

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

[12]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.

Finally, this minimum is fed to the *raising* function f^R together with the old value for x (and possibly any other inputs $\bar{e}$), obtaining a result that is higher than at least one of the two parameters. We assume that the second partial order $\lhd$ is *noetherian*,[13] 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 f^{MP} along every possible path (and the effect of the raising function can be proved to be negligible).

Example 16. *Function f3 in Example 9 does not respect the minimising* rep *pattern, since its internal function is monotonic (see Example 11) but not progressive (see Example 12). However, if we change f3 to the following:*

```
def hopcount(v) { rep (v) { (x) => min(minHood(nbr{x}) + 1, v) } }
```

we obtain a hop-count gradient*, a particular case of the* G *block which is self-stabilising and complies with the minimising* rep *pattern.*

Note that the well-foundedness and noetheriality 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 Appendix A.1, 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.[14] Then s is self-stabilising.*

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:*

- *Each* rep *in a self-stabilising fragment self-stabilises to the same value under arbitrary substitution of the initial condition.*

- *The* converging rep *pattern self-stabilises to the same value as the single expression s occurring in it.*

- *The* minimising rep *pattern self-stabilises to the same value as the analogous pattern where f^R is the identity on its first argument.*

---

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

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

In other words, the function 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 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 3.4 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:

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

Furthermore, the present fragment strictly includes the one defined in [49]. Both fragments include all expressions without the `rep` construct. The first and third `rep` pattern in [49] are special cases of *converging* `rep` (the first converges to the value v_0 in the *bounded* condition and the third to the value 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 could be obtained by the *replicated gossip* [43] 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 [49]). 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 [44].[15]

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 [5]. 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 [45]):

```
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 [4] (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) {
```

---

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

```
          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 6 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.

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[16] 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))
```

---

[16]A structure $\langle X, \circ \rangle$ is an abelian monoid if $\circ$ is an associative and commutative operator with identity.
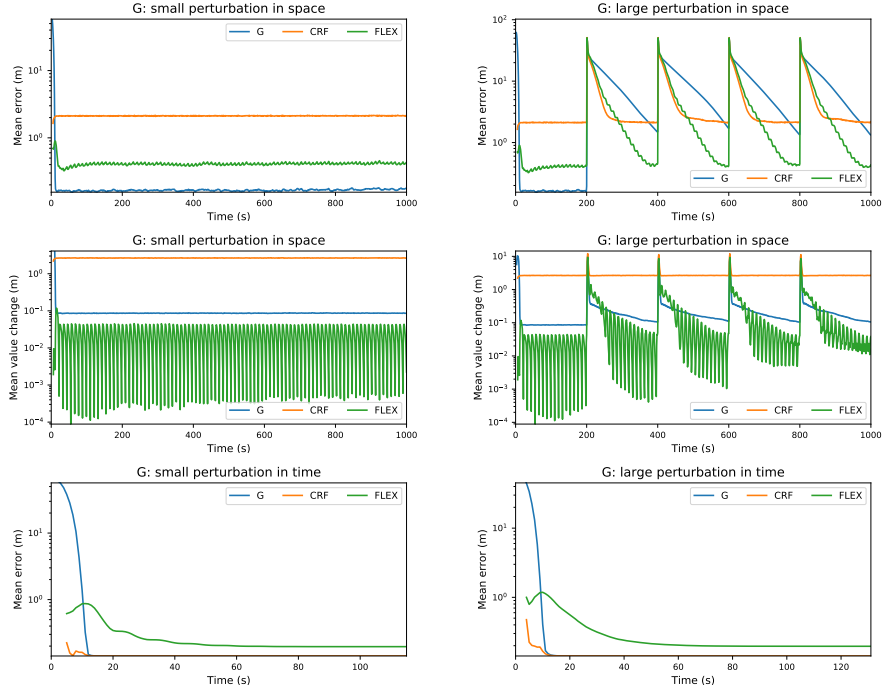
Figure 6: Evaluation of G building blocks: plain G (blue), CRF (green) and FLEX (red). We measure the average error across all devices (first and last row) and the stability of the value, namely, the average value change between subsequent rounds (middle row). With small spatial perturbations, G provides 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.

```
}

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

def C'(potential, local, null)(accumulate, root) {
  rep ( pair(local,local) ) { (x) =>
    aggregate(mux(nbr{potential} < potential, nbr{x}, null),
      local, potential)(accumulate, root)
  }
}
```

We evaluate the multi-path alternative of C when used to sum values of a field, using the following variation of `C_sum`[17]:

```
def C'_sum(potential, field) {  1st( C'(potential, value, 0)(+, /)) }
```

---

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

33

Figure 7: 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.

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 7 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.

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.

34

```
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)
  }
}
```

We evaluate the use of T' in tracking a noisy signal, using the following variation of `T_track`

```
def T'_track(value) {  T'(value,value)(meanHood, x => a*x)}
```

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 8 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)))
}
```
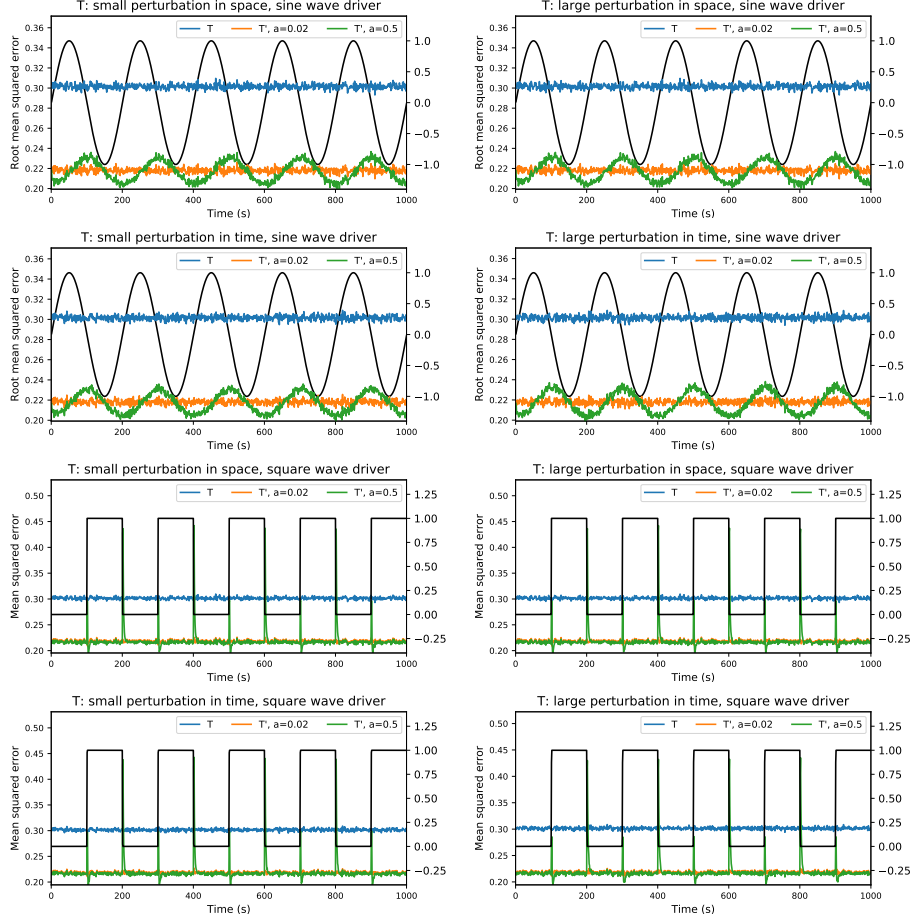
Figure 8: Evaluation of T (blue) and T' building blocks with different smoothing parameter values (`a` = 0.02 in green, and `a` = 0.5 in red). The driver signal (plotted in black for reference) is locally summed with a random noise in $[-1, 1]^{10}$ and fed to the algorithm for tracking. We measure the 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.

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 9. Devices run asynchronously, performing a round of compu-

Figure 9: 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.

tation and communication every five seconds, and communicating by broadcast within a radius of 150 metres (ignoring buildings and other physical obstacles). Our implementation is realised in Protelis [45] and simulations were performed using Alchemist [44]. 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.

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:

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

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 10 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 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.
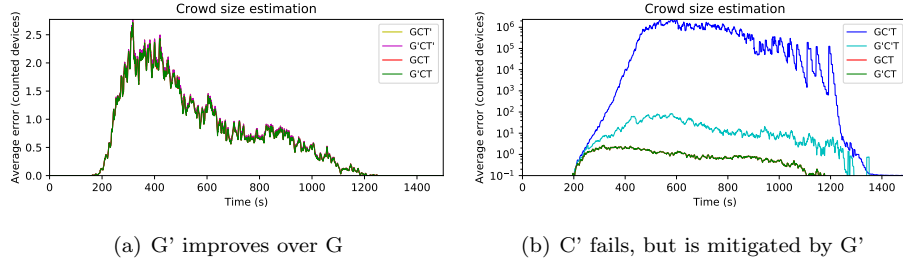
(a) G' improves over G

(b) C' fails, but is mitigated by G'

Figure 10: 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.

Figure 11: 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.

```
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 11.
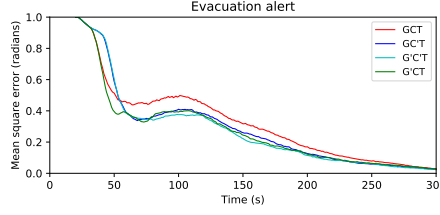
Figure 12: Results for the evacuation alert scenario: G' and C' both improve performance significantly over G and C, respectively. Additional incremental improvement can be obtained by using both G' and C'. The choice of T versus T' has no significant effect on performance.

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 verctor 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:

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

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 12 shows the results, averaged 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 will be best served by different combinations and tradeoffs in the dynamics of building block implementations: for example, choosing G' over G helped in both application scenarios, while C' is useful in the second but not the first, and neither had noisy enough changes for T' to be significantly beneficial over T. The approach that we have implemented allows such combinations to be rapidly and safely explored, enabling optimisation of distributed systems without their re-design.

8   Conclusions

Using the 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 a set of general "building block" operators that conceptually simplify the specification of programs within this class. Such a class is formalised in terms of a fragment of the field calculus, closed under composition, and flexible enough to also include various alternative implementations of the building blocks, allowing dynamical performance to be optimised while still guaranteed to converge 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 for improvement is to obtain a more detailed characterisation for the dynamic trade-space, in order to enable a more systematic approach to optimisation via mechanism substitution. In addition to making human engineering easier, this may also enable automated substitution optimisation, both during the engineering process and dynamically at run-time. Furthermore, alternative definitions of *self-stabilisation* could be inspected, for capturing and describing wider classes of resilient program behaviours (such as replicated gossip [43]) or for allowing a better modeling of important aspects of spatial computations (such as space-time information), as well as integration with dynamical response models such as those presented in [21, 39]. Other important directions for improvement are 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.

Appendix

A.1 Proof of self-stabilisation for the fragment

In this appendix we report complete proofs for the statements given in Section 5.3. We first prove self-stabilisation for the minimising `rep` pattern (Lemma 3), since it is technically more involved than the proof of self-stabilisation for the remainder of the fragment. We then prove self-stabilisation through a variation of the goal results (Lemma 4) more suited for inductive reasoning. Theorems 1 and 2 will then follow by inspecting the proof of those lemmas.

Let $\mathtt{s_{min}} = \mathtt{rep(e)\{(x)=>f^R(minHoodLoc(f^{MP}(nbr\{x\},\overline{s}),s),x,\overline{e})\}}$ be a minimising `rep` expression such that $[\![\overline{\mathtt{s}}]\!] = \overline{\Phi}$, $[\![\mathtt{s}]\!] = \Phi$. Let $P = \overline{\delta}$ be a path in the network (a sequence of pairwise connected devices), and define its *weight* in $\mathtt{s_{min}}$ as the result of picking the eventual value $\ell_1 = \Phi(\delta_1)$ of $\mathtt{s}$ in the first device δ_1, and repeatedly passing it to subsequent devices through the monotonic progressive function, so that $\ell_{i+1} = \mathtt{f^{MP}}(\ell_i, \overline{\mathtt{v}})$ where $\overline{\mathtt{v}}$ is the result of projecting fields in $\overline{\Phi}(\delta_{i+1})$ to their δ_i component (leaving local values untouched). Notice that the weight is well-defined since function $\mathtt{f^{MP}}$ is required to be stateless.

Lemma 3. *Let $\mathtt{s}$ be a minimising `rep` expression. Then $\mathtt{s}$ self-stabilises in each device δ to the minimal weight in $\mathtt{s}$ for a path P ending in δ.*

Proof. Let ℓ_δ be the minimal weight for a path P ending in δ, and let $\delta^0, \delta^1, \ldots$ be the list of all devices δ ordered by increasing ℓ_δ. Notice that the path P of minimal weight ℓ_{δ^i} for device i can only pass through nodes such that $\ell_{\delta^j} \leq \ell_{\delta^i}$ (thus s.t. $j < i$). In fact, whenever a path P contains a node j the weight of its prefix until j is at least ℓ_{δ^j}; thus any longer prefix has weight strictly greater than ℓ_{δ^j} since $\mathtt{f^{MP}}$ is progressive.

Let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$ be a fair evolution[18] and assume w.l.o.g. that all subexpressions of $\mathtt{s}$ not involving $\mathtt{x}$ have already self-stabilised to computational fields $\overline{\Phi}$, Φ (as in the definition of weight) in the initial state N_0. We now prove by complete induction on i that device δ^i stabilises to ℓ_{δ^i} after a certain step t_i.

Assume that devices δ^j with $j < i$ are all self-stabilised (from a certain step t_{i-1}), and consider the evaluation of expression $\mathtt{s}$ in a device δ^k with $k \geq i$. Since the local argument ℓ of $\mathtt{minHoodLoc}$ is also the weight of the single-node path $P = \delta^k$, it has to be at least $\ell \geq \ell_{\delta^k} \geq \ell_{\delta^i}$. Similarly, the restriction ϕ' of the field argument ϕ of $\mathtt{minHoodLoc}$ to devices δ^j with $j < i$ has to be at least $\phi' \geq \ell_{\delta^k} \geq \ell_{\delta^i}$ since it corresponds to weights of (not necessarily minimal) paths P ending in δ^k (obtained by extending a minimal path for a device δ^j with $j < i$ with the additional node δ^k). Finally, the complementary restriction ϕ'' of ϕ to devices δ^j with $j \geq i$ is strictly greater than the minimum value for $\mathtt{x}$ among those devices, since $\mathtt{f^{MP}}$ is progressive.

It follows that as long as the minimum value for $\mathtt{x}$ among non-stable devices is lower than ℓ_{δ^i}, the result of the $\mathtt{minHoodLoc}$ subexpression is strictly greater than this minimum value. Since the overall value of $\mathtt{s}$ is obtained by combining the output of $\mathtt{minHoodLoc}$ with the previous value for $\mathtt{x}$ through the rising function $\mathtt{f^R}$ (and a rising function does not drop below the minimum of its arguments), the minimum value for $\mathtt{x}$ among non-stable devices cannot decrease as long as it is lower than ℓ_{δ^i}, and it cannot drop below ℓ_{δ^i} if it is already greater than that.

---

[18]Notice that δ_0 is the first device firing while δ^0 is the device with minimal weight.

Furthermore, the minimum has to eventually increase until it reaches at least ℓ_{δ^i}. Recall that a rising function selects its first argument infinitely often (since the order $\lhd$ is noetherian). Thus each device realising a minimum for $\mathtt{x}$ among non-stable devices has to eventually evaluate $\mathtt{s}$ to the output of the $\mathtt{minHoodLoc}$ subexpression, which is strictly higher than the previous minimum, and it will not be able to reach the previous minimum afterwards.

Let $t' \geq t_{i-1}$ be the first step in which the minimum for $\mathtt{x}$ among non-stable devices is at least ℓ_{δ^i}, and consider device δ^i. Let P be a path of minimum weight for δ^i, then either:

- $P = \delta^i$, so that ℓ_{δ^i} is exactly the local argument of the $\mathtt{minHoodLoc}$ operator, hence also the output of it (since the field argument is greater than ℓ_{δ^i}).

- $P = Q, \delta^i$ where Q ends in δ^j with $j < i$. Since $\mathtt{f}^{\mathsf{MP}}$ is monotonic non-decreasing, the weight of Q', δ^i (where Q' is minimal for δ^j) is not greater than that of P; in other words, $P' = Q', \delta^i$ is also a path of minimum weight. It follows that $\phi(\delta^j)$ (where ϕ is the field argument of the $\mathtt{minHoodLoc}$ operator) is exactly ℓ_{δ^i}.

In both cases, the output of $\mathtt{minHoodLoc}$ in δ^i stabilises to ℓ_{δ^i} from t' on. Let t_i be the first step after t' in which the rising function $\mathtt{f}^{\mathsf{R}}$ selects its first argument ℓ_{δ^i}. Then expression $\mathtt{s}$ in device δ^i is self-stabilised to ℓ_{δ^i} from t_i on, concluding the inductive step and the proof. $\qquad\square$

Let Φ be a computational field as defined in Section 4.2. We write $\mathtt{s}[\mathtt{x} := \Phi]$ to indicate an aggregate process in which each device is computing a possibly different substitution $\mathtt{s}[\mathtt{x} := \Phi(\delta)]$ of the same expression.

Lemma 4. *Assume that every built-in operator is self-stabilising. Let $\mathtt{s}$ be an expression with free variables $\overline{\mathtt{x}}$ in the self-stabilising fragment, and $\overline{\Phi}$ be a sequence of computational fields of the same length. Then $\mathtt{s}[\overline{\mathtt{x}} := \overline{\Phi}]$ is self-stabilising.*

Proof. The proof proceeds by induction on the syntax of expressions and programs. Let $\mathtt{s}$ be an expression in the fragment, then it can be:

- A variable $\mathtt{x}_i$, so that $\mathtt{s}[\overline{\mathtt{x}} := \overline{\Phi}] = \Phi_i$ is already self-stabilised.

- A value $\mathtt{v}$, so that $\mathtt{s}[\overline{\mathtt{x}} := \overline{\Phi}] = \mathtt{v}$ is already self-stabilised.

- A $\mathtt{let}$-expression $\mathtt{let}\ \mathtt{x} = \mathtt{s}_1\ \mathtt{in}\ \mathtt{s}_2$. Fix an environment *Env*, in which expression $\mathtt{s}_1$ self-stabilises to Φ after fire t. After t, $\mathtt{let}\ \mathtt{x} = \mathtt{s}_1\ \mathtt{in}\ \mathtt{s}_2$ evaluates to the same value of the expression $\mathtt{s}_2[\mathtt{x} := \Phi]$ which is self-stabilising by inductive hypothesis.

- A functional application $\mathtt{f}(\overline{\mathtt{s}})$. Fix an environment *Env*, in which all expressions $\overline{\mathtt{s}}$ self-stabilise to $\overline{\Phi}$ after fire t. After t, if $\mathtt{f}$ is a built-in function then $\mathtt{f}(\overline{\mathtt{s}})$ is already self-stabilised. Otherwise, if $\mathtt{f}$ is a user-defined function then $\mathtt{f}(\overline{\mathtt{s}})$ evaluates to the same value of the expression $body(\mathtt{f})[args(\mathtt{f}) := \overline{\Phi}]$ which is self-stabilising by inductive hypothesis.

- A conditional $\mathtt{s} = \mathtt{if(s_1)\{s_2\}\{s_3\}}$. Fix an environment Env, in which expression $\mathtt{s_1}$ self-stabilises to Φ_{guard}. Let $Env_{\mathtt{True}}$ be the sub-environment consisting of devices δ such that $\Phi_{guard}(\delta) = \mathtt{True}$, and analogously $Env_{\mathtt{False}}$. Assume that $\mathtt{s_2}$ self-stabilises to $\Phi_{\mathtt{True}}$ in $Env_{\mathtt{True}}$ and $\mathtt{s_3}$ to $\Phi_{\mathtt{False}}$ in $Env_{\mathtt{False}}$. Since a conditional is computed in isolation in the above defined sub-environments, $\mathtt{s}$ self-stabilises to $\Phi = \Phi_{\mathtt{True}} \cup \Phi_{\mathtt{False}}$.

- A neighbourhood field construction $\mathtt{nbr\{s\}}$. Fix an environment Env, in which expression $\mathtt{s}$ self-stabilises to Φ after fire t. Then $\mathtt{nbr\{s\}}$ self-stabilises to the corresponding Φ' after one more firing of each device, where $\Phi'(\delta)$ is Φ restricted to $\tau(\delta)$.

- A converging $\mathtt{rep}$: $\mathtt{s} = \mathtt{rep(e)\{(x)\texttt{=>}f^C(nbr\{x\},nbr\{s\},\overline{e})\}}$. Fix an environment Env and a fair evolution of the network $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$, and let t be such that all subexpressions of $\mathtt{s}$ not containing $\mathtt{x}$ have self-stabilised after t. Assume that $\mathtt{s}$ self-stabilises to Φ; we prove that $\mathtt{s}$ stabilises as well to the same Φ.

  Given any index $i \geq t$, let d^i be the maximum distance $\mathtt{x} - \Phi(\delta^i)$ of $\mathtt{x}$ from $\mathtt{s}$ realised by a device δ^i in the network. Let $t_0 = t$ and t_{i+1} be the first firing of device δ^{t_i} after t_i. Since δ^{t_i} realises the maximum distance in the whole network N_{t_i}, no device firing between t_i and t_{i+1} can assume a value more distant than d^{t_i} without violating the converging property. Thus d^i, δ^i remains the same in the whole interval from t_i to t_{i+1} (excluded).

  Finally, in fire t_{i+1} device δ^{t_i} recomputes its value, necessarily obtaining a closer value to $\Phi(\delta^{t_i})$ (by the converging property) thus forcing the overall maximal distance in the network to reduce: $d^{t_{i+1}} < d^{t_i}$. Since the set of possible values is finite, so are the possible distances and eventually the maximal distance d^i will reach 0.

- An acyclic $\mathtt{rep}$: $\mathtt{s} = \mathtt{rep(e)\{(x)\texttt{=>}f(mux(nbrlt(s_p),nbr\{x\},s),\overline{s})\}}$. Fix an environment Env and a fair evolution of the network $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$, and let t be such that all subexpressions of $\mathtt{s}$ not containing $\mathtt{x}$ have self-stabilised after t.

  Let $t_0 \geq t$ be any fire of the device δ^0 of minimal potential $\mathtt{s_p}$ in the network after t. Since δ^0 is minimal, $\mathtt{mux(nbrlt(s_p),nbr\{x\},s)}$ reduces to $\mathtt{s}$ and the whole $\mathtt{s}$ to $\mathtt{f(s,\overline{s})}$, which is self-stabilising (after some $t_0' \geq t_0$) for inductive hypothesis.

  Let $t_1 \geq t_0'$ be any fire of the device δ^1 of second minimal potential after t_0'. Then $\mathtt{mux(nbrlt(s_p),nbr\{x\},s)}$ in δ^1 only (possibly) depends on the value of the device of minimal potential, which is already self-stabilised. Thus by inductive hypothesis $\mathtt{s}$ self-stabilises also in δ^1 after some index $t_1' \geq t_1$. By repeating the same reasoning on all devices in order of increasing potential, we obtain a final t_n' after which all devices have self-stabilised.

- A minimising $\mathtt{rep}$: this case is proved for closed expressions in Lemma 3, and its generalisation to open expressions is straightforward.

$\square$

Theorem 5 (Restatement of 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. Then* s *is self-stabilising.*

Proof. Follows directly from Lemma 4 when s has no free variables. □

Theorem 6 (Restatement of Theorem 2 (Substitutability))**.** *The following three equivalences hold:*

- *Each* rep *in a self-stabilising fragment self-stabilises to the same value under arbitrary substitution of the initial condition.*

- *The* converging rep *pattern self-stabilises to the same value of the single expression* s *occurring in it.*

- *The* minimising rep *pattern self-stabilises to the same value of the analogous pattern where* f^R *is the identity on its first argument.*

Proof. Follows from inspecting the proof of Lemmas 3 and 4. □

References

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

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

[3] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991.

[4] Jacob Beal. Flexible self-healing gradients. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1197–1201, New York, NY, USA, 2009. ACM.

[5] Jacob Beal, Jonathan Bachrach, Daniel Vickery, and Mark Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975. ACM, 2008.

[6] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`.

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

[8] Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. In *2nd FoCAS Workshop on Fundamentals of Collective Systems*, pages 1–6. IEEE CS, to appear, 2014.

[9] Jacob Beal and Mirko Viroli. *Aggregate Programming: From Foundations to Applications*, pages 233–260. Springer International Publishing, Cham, 2016.

[10] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution changes. In Giacomo Cabri, Gauthier Picard, and Niranjan Suri, editors, *10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2016, Augsburg, Germany, September 12-16, 2016*, pages 60–69, 2016. Best paper of IEEE SASO 2016.

[11] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, USA, 2002.

[12] Roberto Casadei and Mirko Viroli. Towards aggregate programming in scala. In *First Workshop on Programming Models and Languages for Distributed Computing*, PMLDC '16, pages 5:1–5:7, New York, NY, USA, 2016. ACM.

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

[14] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. PhD thesis, MIT, Cambridge, MA, USA, 1999.

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

[16] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982.

[17] Ferruccio Damiani and Mirko Viroli. Type-based self-stabilisation for computational fields. *Logical Methods in Computer Science*, 11(4), 2015.

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

[19] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer International Publishing, 2015.

[20] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*, pages 22:1–22:10. IEEE, 2013.

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

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] EW Dijkstra. Ewd391 self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*, pages 41–46. Springer-Verlag, 1982. EWD391's original date is 1973.

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

[25] Shlomi Dolev and Ted Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 1997.

[26] Bradley R. Engstrom and Peter R. Cappello. The sdef programming system. *Journal of Parallel and Distributed Computing*, 7(2):201 – 231, 1989.

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

[28] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computations in space and space in computations. In Jean-Pierre Bantre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 137–152. Springer Berlin Heidelberg, Berlin, 2005.

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

[30] Ted Herman and Imran Pirwani. A composite stabilizing data structure. In *WSS01 Proceedings of the Fifth International Workshop on Self-Stabilizing Systems, Springer LNCS:2194*, pages 167–182, 2001.

[31] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

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

[33] Attila Kondacs. Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

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

[35] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *CoRR*, abs/1610.00253, 2016.

[36] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, December 2002.

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

[38] Renato E. Mirollo and Steven H. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, 1990.

[39] Yuanqiu Mo, Jacob Beal, and Soura Dasgupta. Error in self-stabilizing spanning-tree estimation of collective state. In *Workshop eCAS in Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2017 IEEE International Conference on.* IEEE, Sept 2017.

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

[41] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks*, pages 78–87, aug 2004.

[42] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, January 2007.

[43] Danilo Pianini, Jacob Beal, and Mirko Viroli. Improving gossip dynamics through overlapping replicates. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2016.

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

[45] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *ACM Symposium on Applied Computing 2015*, pages 1846–1853, April 2015.

[46] F Raimbault and D Lavenier. Relacs for systolic programming. In *Int'l Conf. on Application-Specific Array Processors*, pages 132–135, October 1993.

[47] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.

[48] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[49] Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Efficient engineering of complex self-organising systems by self-stabilising fields. In *Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th International Conference on*, pages 81–90. IEEE, Sept 2015.

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

[51] Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In *Coordination Languages and Models*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag, jun 2014.

[52] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.

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

[54] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.

[55] Daniel Yamins. *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*. PhD thesis, Harvard, Cambridge, MA, USA, December 2007.

[56] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31:2002, 2002.

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