

FIELD-BASED COORDINATION WITH THE SHARE OPERATOR

GIORGIO AUDRITO ^a, JACOB BEAL ^b, FERRUCCIO DAMIANI ^a, DANILO PIANINI ^c,
AND MIRKO VIROLI ^c

^a Dipartimento di Informatica, University of Torino, Torino, Italy
e-mail address: giorgio.audrito@unito.it
e-mail address: ferruccio.damiani@unito.it

^b Raytheon BBN Technologies, Cambridge (MA), USA
e-mail address: jakebeal@ieee.org

^c Alma Mater Studiorum–Università di Bologna, Italy
e-mail address: danilo.pianini@unibo.it
e-mail address: mirko.viroli@unibo.it

ABSTRACT. Field-based coordination has been proposed as a model for coordinating collective adaptive systems, promoting a view of distributed computations as functions manipulating data structures spread over space and evolving over time, called computational fields. The field calculus is a formal foundation for field computations, providing specific constructs for evolution (time) and neighbour interaction (space), which are handled by separate operators (called **rep** and **nbr**, respectively). This approach, however, intrinsically limits the speed of information propagation that can be achieved by their combined use. In this paper, we propose a new field-based coordination operator called **share**, which captures the space-time nature of field computations in a single operator that declaratively achieves: (i) observation of neighbours' values; (ii) reduction to a single local value; and (iii) update and converse sharing to neighbours of a local variable. We show that for an important class of self-stabilising computations, **share** can replace all occurrences of **rep** and **nbr** constructs. In addition to conceptual economy, use of the **share** operator also allows many prior field calculus algorithms to be greatly accelerated, which we validate empirically with simulations of frequently used network propagation and collection algorithms.

Key words and phrases: Aggregate computing, field calculus, information propagation.

This work has been partially supported by Ateneo/CSP project “AP: Aggregate Programming” (<http://ap-project.di.unito.it/>) and by Italian PRIN 2017 project “Fluidware”. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations.

1. INTRODUCTION

The number and density of networking computing devices distributed throughout our environment is continuing to increase rapidly. In order to manage and make effective use of such systems, there is likewise an increasing need for software engineering paradigms that simplify the engineering of resilient distributed systems. Aggregate programming [BPV15, VBD⁺19] is one such promising approach, providing a layered architecture in which programmers can describe computations in terms of resilient operations on “aggregate” data structures with values spread over space and evolving in time.

The foundation of this approach is field computation, formalized by the field calculus [VAB⁺18], a terse mathematical model of distributed computation that simultaneously describes both collective system behavior and the independent, unsynchronized actions of individual devices that will produce that collective behavior [AVD⁺19]. In this approach each construct and reusable component is a pure function from fields to fields—a field is a map from a set of space-time computational events to a set of values. In prior formulations, each primitive construct has also handled just one key aspect of computation: hence, one construct deals with time (i.e., **rep**, providing field evolution, in the form of periodic state updates) and one with space (i.e., **nbr**, handling neighbour interaction, in the form of reciprocal state sharing).

However, in recent work on the universality of the field calculus, we have identified that the combination of time evolution and neighbour interaction operators in the original field calculus induces a delay, limiting the speed of information propagation that can be achieved efficiently [ABDV18]. This limit is caused by the separation of state sharing (**nbr**) and state updates (**rep**), which means that any information received with a **nbr** operation has to be remembered with a **rep** before it can be shared onward during the next execution of the **nbr** operation, as illustrated in Figure 1.

In this paper, we address this limitation by extending the field calculus with the **share** construct. Building on the underlying asynchronous protocol of field calculus, this extension combines time evolution and neighbour interaction into a single new atomic coordination primitive that simultaneously implements: (i) observation of neighbours’ values; (ii) reduction to a single local value; and (iii) update of a local variable and sharing of the updated value with neighbours. The **share** construct thus allows the effects of information received from neighbours to be shared immediately after it is incorporated into state, rather than having to wait for the next round of computation.

Another contribution of this paper is the adaptation of the field calculus operational semantics in [VAB⁺18] to model *true concurrency*, i.e., modelling non-instantaneous computation rounds. This extension, required to fully capture the semantics of the **share** construct, is shown to be conservative with respect to [VAB⁺18], and extends the applicability of the calculus by mirroring the denotational semantics [AVD⁺19] (which was already true concurrent) on *augmented event structures* (a novel refined definition capturing physically realisable aggregate computations).

The remainder of this paper formally develops and experimentally validates these concepts, expanding on a prior version [ABD⁺19] with an improved and extended presentation of the operators, complete formal semantics (including the true concurrent version of the network semantics in [VAB⁺18]), analysis of key properties, and additional experimental validation. Following a review of the field calculus and its motivating context in Section 2, we introduce the novel network semantics in Section 3, and the **share** construct in Section 4,

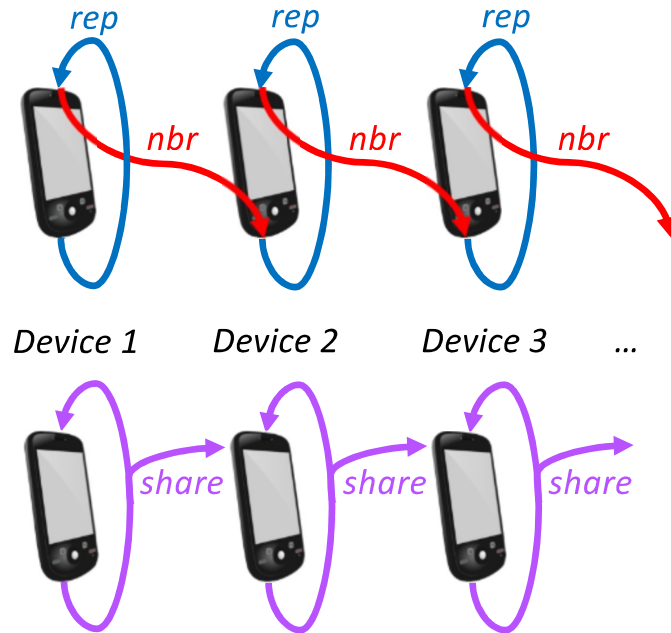


Figure 1: Handling state sharing (*nbr*) and memory (*rep*) separately injects a delay while information “loops around” to where it can be shared (top), while combining state sharing and memory into the new *share* operator eliminates that delay (bottom).

along with formal semantics and analysis of the relationship of the *share* construct with the combined use of the *rep* and *nbr* constructs. We then empirically validate the predicted acceleration of speed in frequently used network propagation and collection algorithms in Section 5, and conclude with a summary and discussion of future work in Section 6.

2. RELATED WORK AND BACKGROUND

Programming collective adaptive systems is a challenge that has been recognized and addressed in a wide variety of different contexts. Despite the wide variety of goals and starting points, however, the commonalities in underlying challenges have tended to shape the resulting aggregate programming approaches into several clusters of common approaches, as enumerated in [BDU⁺13]: *(i)* “device-abstraction” methods that abstract and simplify the programming of individual devices and interactions (e.g., TOTA [MZ09], Hood [WSBC04], chemical models [VPM⁺15], “paintable computing” [But02], Meld [ARGL⁺07]) or entirely abstract away the network (e.g., BSP [Val90], MapReduce [DG08], Kairos [GGG05]); *(ii)* spatial patterning languages that focus on geometric or topological constructs (e.g., Growing Point Language [Coo99], Origami Shape Language [Nag01], self-healing geometries [CN03, Kon03], cellular automata patterning [Yam07]); *(iii)* information summarization languages that focus on collection and routing of information (e.g., TinyDB [MFHH02], Cougar [YG02], TinyLime [CGG⁺05], and Regiment [NW04]); *(iv)* general purpose space-time computing models (e.g., StarLisp [LMMD88], MGS [GGMP02, GMCS05], Proto [BB06], aggregate programming [BPV15]).

The field calculus [VAB⁺18, AVD⁺19] belongs to the last of these classes, the general purpose models. Like other core calculi, such as λ -calculus [Chu32] or Featherweight Java [IPW01], the field calculus provides a minimal, mathematically tractable programming language—in this case with the goal of unifying across a broad class of aggregate programming approaches and providing a principled basis for integration and composition. Indeed, recent analysis [ABDV18] has determined that the current formulation of field calculus is space-time universal, meaning that it is able to capture every possible computation over collections of devices sending messages. Field calculus can thus serve as a unifying abstraction for programming collective adaptive systems, and results regarding field calculus have potential implications for all other works in this field. Indeed, all of the algorithms we discuss in this paper are generalized versions that unify across the common patterns found in all of the works cited above, as described in [BDU⁺13, FMSM⁺13, VAB⁺18].

In addition to establishing universality, however, the work in [ABDV18] also identified a key limitation of the current formulation of the field calculus, which we are addressing in this paper. In particular, the operators for time evolution and neighbour interaction in field calculus interact such that for most programs either the message size grows with the distance that information must travel or else information must travel significantly slower than the maximum potential speed. The remainder of this section provides a brief review of these key results: Section 2.1 introduces the underlying space-time computational model used by the field calculus (featuring a novel refined definition of *augmented event structure* capturing the physically realisable aggregate computations), Section 2.2 introduces the notion of self-stabilisation, Section 2.3 provides a review of the field calculus itself, followed by a review of its device semantics (modeling the local and asynchronous computation that takes place on a single device) in Section 2.4. The network semantics (modeling the overall network evolution) will then be presented in Section 3.

2.1. Space-Time Computation. Field calculus considers a computational model in which a program P is periodically and asynchronously executed by each device δ .¹ When an individual device performs a round of execution, that device follows these steps in order: (i) collects information from sensors, local memory, and the most recent messages from neighbours,² the latter organised into *neighbouring value maps* $\phi : \delta \rightarrow \mathbf{v}$ from neighbour identifiers to neighbour values, (ii) evaluates program P with the information collected as its input, (iii) stores the results of the computation locally, as well as broadcasting it to neighbours and possibly feeding it to actuators, and (iv) sleeps until it is time for the next round of execution. Note that as execution is asynchronous, devices perform executions independently and without reference to the executions of other devices, except insofar as they use state that has arrived in messages. Messages, in turn, are assumed to be collected by some separate thread, independent of execution rounds. Note that the `share` operator we discuss in this paper works on top of the above execution model, hence it affects the local evaluation of the program, which in turn results in the exchange of asynchronous messages.

If we take every such execution as an *event* ϵ , then the collection of such executions across space (i.e., across devices) and time (i.e., over multiple rounds) may be considered as the execution of a single aggregate machine with a topology based on information exchanges \rightsquigarrow . The causal relationship between events may then be formalized as defined in [Lam78]:

¹We use δ as a metavariable ranging over a given denumerable set of device identifiers D .

²Stale messages may expire after some timeout.

Definition 2.1 (Event Structure). An *event structure* $\langle E, \rightsquigarrow, < \rangle$ is a countable set of *events* E together with a neighbouring relation $\rightsquigarrow \subseteq E \times E$ and a causality relation $< \subseteq E \times E$, such that the transitive closure of \rightsquigarrow forms the irreflexive partial order $<$, and the set $X_\epsilon = \{\epsilon' \in E \mid \epsilon' < \epsilon\} \cup \{\epsilon' \in E \mid \epsilon \rightsquigarrow \epsilon'\}$ is finite for all ϵ (i.e., $<$ and \rightsquigarrow are locally finite).

Thus, we say that ϵ' is a neighbour of ϵ iff $\epsilon' \rightsquigarrow \epsilon$, and that $\mathcal{N}(\epsilon) = \{\epsilon' \in E \mid \epsilon' \rightsquigarrow \epsilon\}$ is the set of neighbours of ϵ .

Remark 2.2 (Event Structures and Petri Nets). Event structures for Petri Nets are used to model a spectrum of *possible evolutions* of a system, hence include also an *incompatibility* relation, discriminating between alternate future histories and modelling non-deterministic choice. However, following [Lam78], we use event structures to model a “timeless” *unitary history* of events, thus avoiding the need for an incompatibility relation.

In aggregate computing, event structures need to be *augmented* with device identifiers [AVD⁺19, ABDV18], as in the following definition.

Definition 2.3 (Augmented Event Structure). Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be such that $\langle E, \rightsquigarrow, < \rangle$ is an event structure and $d : E \rightarrow D$ is a mapping from events to the devices where they happened. We define:

- $\text{next} : E \rightarrow E$ as the partial function³ mapping an event ϵ to the unique event $\text{next}(\epsilon)$ such that $\epsilon \rightsquigarrow \text{next}(\epsilon)$ and $d(\epsilon) = d(\text{next}(\epsilon))$, if such an event exists and is unique; and
- $\dashrightarrow \subseteq E \times E$ as the relation such that $\epsilon \dashrightarrow \epsilon'$ (ϵ *implicitly precedes* ϵ') if and only if $\epsilon' \rightsquigarrow \text{next}(\epsilon)$ and $\epsilon' \not\rightsquigarrow \epsilon$.

We say that \mathbf{E} is an *augmented event structure* if the following coherence constraints are satisfied:

- **linearity:** if $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ and $d(\epsilon) = d(\epsilon_1) = d(\epsilon_2)$, then $\epsilon_1 = \epsilon_2 = \text{next}(\epsilon)$ (i.e., every event ϵ is a neighbour of at most another one on the same device);
- **uniqueness:** if $\epsilon_i \rightsquigarrow \epsilon$ for $i = 1, 2$ and $d(\epsilon_1) = d(\epsilon_2)$, then $\epsilon_1 = \epsilon_2$ (i.e., neighbours of an event all happened on different devices);
- **impersistence:** if $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ and $d(\epsilon_1) = d(\epsilon_2) = \delta$, then either $\epsilon_2 = \text{next}^n(\epsilon_1)$ and $\epsilon \rightsquigarrow \text{next}^k(\epsilon_1)$ for all $k \leq n$, or the same happens swapping ϵ_1 with ϵ_2 (i.e., an event reaches a contiguous set of events on a same device);
- **immediacy:** there is no cyclic sequence such that $\epsilon_1 < \epsilon_2 \dashrightarrow \epsilon_3 < \dots < \epsilon_{2n} \dashrightarrow \epsilon_1$ (i.e., explicit causal dependencies $<$ are consistent with implicit time dependencies \dashrightarrow).

The first two constraints are necessary for defining the semantics of an aggregate program (denotational semantics in [AVD⁺19, VBD⁺19]). The third reflects that messages are not retrieved after they are first dropped (and in particular, they are all dropped on device reboots). The last constraint reflects the assumption that communication happens through broadcast (modeled as happening instantaneously). In this scenario, the explicit causal dependencies imply additional time dependencies $\epsilon \dashrightarrow \epsilon'$: if ϵ' was able to reach $\text{next}(\epsilon)$ but not ϵ , the broadcast of ϵ' must have happened *after* the start of ϵ (additional details on this point may be found in the proof of Theorem 3.5 in Appendix A).

Remark 2.4 (On Augmented Event Structures). Augmented event structures were first implicitly used in [AVD⁺19] for defining the denotational semantics (with the *linearity* and *uniqueness* constraints only), then formalised in [ABDV18] (without any explicit constraint embedded in the definition). In this paper, we gathered all necessary constraints to capture

³With $A \rightarrow B$ we denote the space of partial functions from A into B .

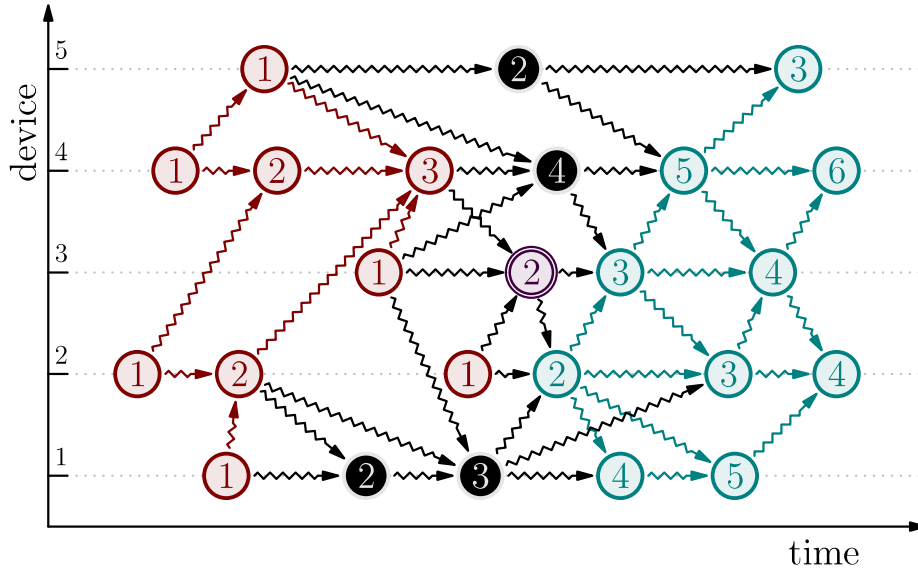


Figure 2: Example of a space-time augmented event structure, comprising events (circles), neighbour relations (arrows), devices (ordinate axis). Colors indicate causal structure with respect to the doubly-circled event (magenta), splitting events into causal past (red), causal future (cyan) and concurrent (non-ordered, in black). The numbers written within events represent a sample space-time value (cf. Def. 2.5) associated with that event structure. Note that the doubly-circled event has 3 neighbouring events: event 1 at the same device (its previous round), event 3 at device 4, and event 1 at device 2. Figure adapted from [ABDV18].

exactly which augmented event structures correspond to physically plausible executions of an aggregate system (see Theorem 3.5): this includes both the *linearity* and *uniqueness* from [AVD⁺19], together with the new *impersistence* and *immediacy* constraints.

Figure 2 shows an example of such an augmented event structure, showing how these relations partition events into “causal past”, “causal future”, and non-ordered “concurrent” subspaces with respect to any given event. Interpreting this in terms of physical devices and message passing, a physical device is instantiated as a chain of events connected by \rightsquigarrow relations (representing evolution of state over time with the device carrying state from one event to the next), and any \rightsquigarrow relation between devices represents information exchange from the tail neighbour to the head neighbour. Notice that this is a very flexible and permissive model: there are no assumptions about synchronization, shared identifiers or clocks, or even regularity of events (though of course these things are not prohibited either).

In principle, an execution at ϵ can depend on information from any event in its past and its results can influence any event in its future. As we will see in Section 4.1, however, this is problematic for the field calculus as it has been previously defined.

Our aggregate constructs then manipulate space-time data values (see Figure 2) that map events to values for each event in an event structure:

Definition 2.5 (Space-Time Value). Let \mathbf{V} be any domain of computational values and $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be an augmented event structure. A space-time value $\Phi = \langle \mathbf{E}, f \rangle$ is a pair

comprising the event structure and a function $f : E \rightarrow \mathbf{V}$ that maps the events $\epsilon \in E$ to values $v \in \mathbf{V}$.

We can then understand an aggregate computer as a “collective” device manipulating such space-time values, and the field calculus as a definition of operations defined both on individual events and simultaneously on aggregate computers, modelled as space-time functions.

Definition 2.6 (Space-Time Function). Let $\mathbf{V}(\mathbf{E}) = \{\langle \mathbf{E}, f \rangle \mid f : E \rightarrow \mathbf{V}\}$ be the set of space-time values in an augmented event structure \mathbf{E} . Then, an n -ary space-time function in \mathbf{E} is a partial map $\mathbf{f} : \mathbf{V}(\mathbf{E})^n \rightarrow \mathbf{V}(\mathbf{E})$.

2.2. Stabilisation and spatial model. Even though the global interpretation of a program has to be given in spatio-temporal terms in general, for a relevant class of programs a space-only representation is also possible. In this representation, *event structures*, *space-time values* and *space-time functions* are replaced by *network graphs*, *computational fields* and *field functions*.

Definition 2.7 (Network Graph). A *network graph* $\mathbf{G} = \langle D, \succ \rangle$ is a finite set D of *devices* δ together with a reflexive neighbouring relation $\succ \subseteq D \times D$, i.e., such that $\delta \succ \delta$ for each $\delta \in D$. Thus, we say that δ' is a neighbour of δ iff $\delta' \succ \delta$, and that $\mathcal{N}(\delta) = \{\delta' \in D \mid \delta' \succ \delta\}$ is the set of neighbours of δ .

Notice that \succ does not necessarily have to be symmetric.

Definition 2.8 (Computational Field). Let \mathbf{V} be any domain of computational values and $\mathbf{G} = \langle D, \succ \rangle$ be a network graph. A *computational field* $\Psi = \langle \mathbf{G}, g \rangle$ is a pair comprising the network graph and a function $g : D \rightarrow \mathbf{V}$ mapping devices $\delta \in D$ to values $v \in \mathbf{V}$.

Definition 2.9 (Field Function). Let $\mathbf{V}(\mathbf{G}) = \{\langle \mathbf{G}, g \rangle \mid g : D \rightarrow \mathbf{V}\}$ be the set of computational fields in a network graph G . Then, an n -ary *field function in G* is a partial map $\mathbf{g} : \mathbf{V}(\mathbf{G})^n \rightarrow \mathbf{V}(\mathbf{G})$.

These space-only, time-independent representations are to be interpreted as “*limits for time going to infinity*” of their traditional time-dependent counterparts, where the limit is defined as in the following.

Definition 2.10 (Stabilising Event Structure and Limit). Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be an infinite augmented event structure. We say that \mathbf{E} is *stabilising* to its limit $\mathbf{G} = \langle D, \succ \rangle = \lim \mathbf{E}$ iff $D = \{\delta \mid \exists^\infty \epsilon \in E. d(\epsilon) = \delta\}$ is the set of devices appearing infinitely often in \mathbf{E} , and for all except finitely many $\epsilon \in E$, the devices of neighbours are the neighbours of the device of ϵ :

$$\{d(\epsilon') \mid \epsilon' \rightsquigarrow \epsilon\} = \{\delta' \mid \delta' \succ d(\epsilon)\}$$

Definition 2.11 (Stabilising Value and Limit). Let $\Phi = \langle \mathbf{E}, f \rangle$ be a space-time value on a stabilising event structure $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ with limit \mathbf{G} . We say that Φ is stabilising to its limit $\Psi = \langle \mathbf{G}, g \rangle = \lim \Phi$ iff for all except finitely many $\epsilon \in E$, $f(\epsilon) = g(d(\epsilon))$.

Definition 2.12 (Self-Stabilising Function and Limit). Let $\mathbf{f} : \mathbf{V}(\mathbf{E})^n \rightarrow \mathbf{V}(\mathbf{E})$ be an n -ary space-time function in a stabilising \mathbf{E} with limit \mathbf{G} . We say that \mathbf{f} is *self-stabilising* with limit $\mathbf{g} : \mathbf{V}(\mathbf{G})^n \rightarrow \mathbf{V}(\mathbf{G})$ iff for any $\langle \Phi_1, \dots, \Phi_n \rangle$ with limit $\langle \Psi_1, \dots, \Psi_n \rangle$, $\mathbf{f}(\Phi_1, \dots, \Phi_n) = \Phi$ with limit $\Psi = \mathbf{g}(\Psi_1, \dots, \Psi_n) = \lim \Phi$.

$P ::= \bar{F} e$	program
$F ::= \text{def } d(\bar{x}) \{e\}$	function declaration
$e ::= x \mid v \mid \text{let } x = e \text{ in } e \mid f(\bar{e}) \mid \text{if}(e)\{e\}\{e\}$ $\quad \mid \text{nbr}\{e\} \mid \text{rep}(e)\{x \Rightarrow e\}$	expression
$f ::= d \mid b$	function name
$v ::= \ell \mid \phi$	value
$\ell ::= c(\bar{\ell})$	local value
$\phi ::= \bar{\delta} \mapsto \bar{\ell}$	neighbouring value

Figure 3: Abstract syntax of the field calculus, adapted from [VAB⁺18]

Many of the most commonly used routines in aggregate computing compute self-stabilising functions, and in fact belong to a self-stabilising class identified in [VAB⁺18]. In Section 4.7, we shall prove that the convergence dynamics of this class can be improved by use of the **share** construct, without changing the overall limit (see Theorem 4.10).

2.3. Field Calculus. The field calculus is a tiny universal language for computation of space-time values. Figure 3 gives an abstract syntax for field calculus based on the presentation in [VAB⁺18] (covering a subset of the higher-order field calculus in [AVD⁺19], but including all of the issues addressed by the **share** construct). In this syntax, the overbar notation \bar{e} indicates a sequence of elements (e.g., \bar{e} stands for e_1, e_2, \dots, e_n), and multiple overbars are expanded together (e.g., $\bar{\delta} \mapsto \bar{\ell}$ stands for $\delta_1 \mapsto \ell_1, \delta_2 \mapsto \ell_2, \dots, \delta_n \mapsto \ell_n$). There are four keywords in this syntax: **def** and **if** respectively correspond to the standard function definition and the branching expression constructs, while **rep** and **nbr** correspond to the two peculiar field calculus constructs that are the focus of this paper, respectively responsible for evolution of state over time and for sharing information between neighbours.

A field calculus program P is a set of function declarations \bar{F} and the main expression e . This main expression e simultaneously defines both the aggregate computation executed on the overall event structure of an aggregate computer and the local computation executed at each of the individual events therein. An expression e can be:

- A *variable* x , e.g. a function parameter.
- A *value* v , which can be of the following two kinds:
 - a *local value* ℓ , defined via data constructor c and arguments $\bar{\ell}$, such as a Boolean, number, string, pair, tuple, etc;
 - A *neighbouring (field) value* ϕ that associates neighbour devices δ to local values ℓ , e.g., a map of neighbours to the distances to those neighbours.
- A *let-expression* **let** $x = e_0$ **in** e , which is evaluated by first computing the value v_0 of e_0 and then yielding 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 $f(\bar{e})$ to either a *user-declared function* d (declared with the **def** keyword) or a *built-in function* b , such as a mathematical or logical operator, a data structure operation, or a function returning the value of a sensor.
- A *branching expression* **if**(e_1){ e_2 } **else** { e_3 }, used to split a computation into operations on two isolated event structures, where/when e_1 evaluates to **true** or **false**: the result is

the local value produced by the computation of e_2 in the former area, and the local value produced by the computation of e_3 in the latter.

- The $\text{nbr}\{e\}$ construct, where e evaluates to a local value, creates a neighbouring value mapping neighbours to their latest available result of evaluating e . In particular, each device δ :
 - (1) shares its value of e with its neighbours, and
 - (2) evaluates the expression into a neighbouring value ϕ mapping each neighbour δ' of δ to the latest value that δ' has shared for e .

Note that within an `if` branch, sharing is restricted to work on device events within the subspace of the branch.

- The $\text{rep}(e_1)\{x \Rightarrow e_2\}$ construct, where e_1 and e_2 evaluate to local values, models state evolution over time: the value of x is initialized to e_1 , then evolved at each execution by evaluating e_2 where x is the result at previous round.

Thus, for example, distance to the closest member of a set of “source” devices can be computed with the following simple function:

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

Here, we use the `def` construct to define a `distanceTo` function that takes a Boolean `source` variable as input. The `rep` construct defines a distance estimate `d` that starts at infinity, then decreases in one of two ways. If the `source` variable is true, then the device is currently a source, and its distance to itself is zero. Otherwise, distance is estimated via the triangle inequality, taking the minimum of a neighbouring value (built-in function `minHood`) of the distance to each neighbour (built-in function `nbrRange`) plus that neighbour’s distance estimate `nbr{d}`. Function `mux` ensures that all its arguments are evaluated before being selected.

2.4. Device Semantics. The local and asynchronous computation that takes place on a single device was formalized in [VAB⁺18] by a big-step semantics, expressed by the judgement $\delta; \Theta; \sigma \vdash e_{\text{main}} \Downarrow \theta$, to be read “expression e_{main} evaluates to θ on device δ with respect to the locally-available environment Θ and locally-available sensor state σ ”. 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 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) through asynchronous message passing. The value-trees recently received as messages from neighbours are then collected into a *value-tree environment* $\bar{\Theta}$, implemented as a map from device identifiers to value-trees (written $\bar{\delta} \mapsto \bar{\theta}$ as short for $\delta_1 \mapsto \theta_1, \dots, \delta_n \mapsto \theta_n$). Intuitively, the outcome of the evaluation will depend on those value-trees. Figure 4 (top) defines value-trees and value-tree environments.

Example 2.13. The graphical representation of the value trees $6\langle 2\langle \rangle, 3\langle \rangle \rangle$ and $6\langle 2\langle \rangle, 3\langle 7\langle \rangle, 1\langle \rangle, 4\langle \rangle \rangle$ is as follows:



In the following, for sake of readability, we sometimes write the value v as short for the value-tree $v\langle \rangle$. Following this convention, the value-tree $6\langle 2\langle \rangle, 3\langle \rangle \rangle$ is shortened to $6\langle 2, 3 \rangle$, and the value-tree $6\langle 2\langle \rangle, 3\langle 7\langle \rangle, 1\langle \rangle, 4\langle \rangle \rangle$ is shortened to $6\langle 2, 3\langle 7, 1, 4 \rangle \rangle$.

Figure 4 (bottom) defines the judgement $\delta; \Theta; \sigma \vdash e \Downarrow \theta$, where: (i) δ is the identifier of the current device; (ii) Θ is the neighbouring value of the value-trees produced by the most recent evaluation of (an expression corresponding to) e on δ 's neighbours; (iii) e is a closed run-time expression (i.e., a closed expression that may contain neighbouring values); (iv) the value-tree θ represents the values computed for all the expressions encountered during the evaluation of e —in particular the root of the value tree θ , denoted by $\rho(\theta)$, is the value computed for expression e . The auxiliary function ρ is defined in Figure 4 (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 e' of 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 4 (second frame). This function has two different behaviors (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 `if` construct). Auxiliary functions ρ and π apply pointwise on value-tree environments, as defined in Figure 4 (second frame, rules for $aux \in \rho, \pi_i, \pi^\ell$).

Rules [E-LOC] and [E-FLD] model the evaluation of expressions that are either a local value or a neighbouring value, respectively: note that in [E-FLD] we take care of restricting the domain of a neighbouring value to the only set of neighbour devices as reported in Θ .

Rule [E-LET] is fairly standard: it first evaluates e_1 and then evaluates the expression obtained from e_2 by replacing all the occurrences of the variable x with the value of e_1 .

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $\mathbf{b}(e_1 \cdots e_n)$, where $n \geq 0$. It produces the value-tree $v\langle \theta_1, \dots, \theta_n \rangle$, where $\theta_1, \dots, \theta_n$ are the value-trees produced by the evaluation of the actual parameters e_1, \dots, e_n and v is the value returned by the function. The rule exploits the special auxiliary function $(\mathbf{b})_\delta^{\Theta, \sigma}$. This function is such that $(\mathbf{b})_\delta^{\Theta, \sigma}(\bar{v})$ computes the result of applying built-in function \mathbf{b} to values \bar{v} in the current environment of the device δ .⁴ In particular: the built-in 0-ary function `self` gets evaluated to the current device identifier (i.e., $(\mathbf{self})_\delta^{\Theta, \sigma}() = \delta$), and mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $(*)_\delta^{\Theta, \sigma}(2, 3) = 6$).

Example 2.14. Evaluating the expression $*(2, 3)$ produces the value-tree $6\langle 2, 3 \rangle$. The value of the whole expression, 6, has been computed by using rule [E-B-APP] to evaluate the application

⁴We do not give the explicit definition of $(\mathbf{b})_\delta^{\Theta, \sigma}(\bar{v})$ for each \mathbf{b} in this paper, and leave it as an implementation detail of the semantics.

Value-trees and value-tree environments:	
$\theta ::= \mathbf{v}(\bar{\theta})$	value-tree
$\Theta ::= \bar{\delta} \mapsto \bar{\theta}$	value-tree environment
<hr/>	
Auxiliary functions:	
$\rho(\mathbf{v}(\bar{\theta})) = \mathbf{v}$	
$\pi_i(\mathbf{v}(\theta_1, \dots, \theta_n)) = \theta_i$ if $1 \leq i \leq n$	$\pi^\ell(\mathbf{v}(\theta_1, \theta_2)) = \theta_2$ if $\rho(\theta_1) = \ell$
$\pi_i(\theta) = \bullet$ otherwise	$\pi^\ell(\theta) = \bullet$ otherwise
For $aux \in \rho, \pi_i, \pi^\ell$: $\left\{ \begin{array}{ll} aux(\delta \mapsto \theta) = \delta \mapsto aux(\theta) & \text{if } aux(\theta) \neq \bullet \\ aux(\delta \mapsto \theta) = \bullet & \text{if } aux(\theta) = \bullet \\ aux(\Theta, \Theta') = aux(\Theta), aux(\Theta') & \end{array} \right.$	
$args(\mathbf{d}) = \bar{x}$ if $\mathbf{def} \mathbf{d}(\bar{x}) \{e\}$	$body(\mathbf{d}) = e$ if $\mathbf{def} \mathbf{d}(\bar{x}) \{e\}$
<hr/>	
Syntactic shorthands:	
$\delta; \bar{\pi}(\Theta); \sigma \vdash \bar{e} \Downarrow \bar{\theta}$ where $ \bar{e} = n$ for $\delta; \pi_1(\Theta); \sigma \vdash e_1 \Downarrow \theta_1 \dots \delta; \pi_n(\Theta); \sigma \vdash e_n \Downarrow \theta_n$	
$\rho(\bar{\theta})$ where $ \bar{\theta} = n$ for $\rho(\theta_1), \dots, \rho(\theta_n)$	
$\bar{x} := \rho(\bar{\theta})$ where $ \bar{x} = n$ for $x_1 := \rho(\theta_1) \dots x_n := \rho(\theta_n)$	
<hr/>	
Rules for expression evaluation: $\delta; \Theta; \sigma \vdash e \Downarrow \theta$	
$\frac{[E-LOC]}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell \langle \rangle}$	$\frac{[E-FLD] \quad \phi' = \phi _{\mathbf{dom}(\Theta) \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi' \langle \rangle}$
$\frac{[E-LET] \quad \delta; \pi_1(\Theta); \sigma \vdash e_1 \Downarrow \theta_1 \quad \delta; \pi_2(\Theta); \sigma \vdash e_2[x := \rho(\theta_1)] \Downarrow \theta_2}{\delta; \Theta; \sigma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 \Downarrow \rho(\theta_2) \langle \theta_1, \theta_2 \rangle}$	
$\frac{[E-B-APP] \quad \delta; \bar{\pi}(\Theta); \sigma \vdash \bar{e} \Downarrow \bar{\theta} \quad \mathbf{v} = (\mathbf{b})_{\delta}^{\Theta, \sigma}(\rho(\bar{\theta}))}{\delta; \Theta; \sigma \vdash \mathbf{b}(\bar{e}) \Downarrow \mathbf{v}(\bar{\theta})}$	
$\frac{[E-D-APP] \quad \delta; \bar{\pi}(\Theta); \sigma \vdash \bar{e} \Downarrow \bar{\theta} \quad \delta; \Theta; \sigma \vdash body(\mathbf{d})[args(\mathbf{d}) := \rho(\bar{\theta})] \Downarrow \theta'}{\delta; \Theta; \sigma \vdash \mathbf{d}(\bar{e}) \Downarrow \rho(\theta') \langle \bar{\theta}, \theta' \rangle}$	
$\frac{[E-NBR] \quad \delta; \pi_1(\Theta); \sigma \vdash e \Downarrow \theta \quad \phi = \rho(\pi_1(\Theta))[\delta \mapsto \rho(\theta)]}{\delta; \Theta; \sigma \vdash \mathbf{nbr}\{e\} \Downarrow \phi \langle \theta \rangle}$	
$\frac{[E-REP] \quad \delta; \pi_1(\Theta); \sigma \vdash e_1 \Downarrow \theta_1 \quad \delta; \pi_2(\Theta); \sigma \vdash e_2[x := \ell_0] \Downarrow \theta_2 \quad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \rho(\theta_1) & \text{otherwise} \end{cases}}{\delta; \Theta; \sigma \vdash \mathbf{rep}(e_1)\{(x) \Rightarrow e_2\} \Downarrow \rho(\theta_2) \langle \theta_1, \theta_2 \rangle}$	
$\frac{[E-IF] \quad \delta; \pi_1(\Theta); \sigma \vdash e \Downarrow \theta_1 \quad \rho(\theta_1) \in \{\mathbf{true}, \mathbf{false}\} \quad \delta; \pi^{\rho(\theta_1)}(\Theta); \sigma \vdash e_{\rho(\theta_1)} \Downarrow \theta}{\delta; \Theta; \sigma \vdash \mathbf{if}(e)\{e_{\mathbf{true}}\}\{e_{\mathbf{false}}\} \Downarrow \rho(\theta) \langle \theta_1, \theta \rangle}$	

Figure 4: Big-step operational semantics for expression evaluation, adapted from [VAB⁺18].

of the multiplication 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 $(\mathbf{b})_{\delta}^{\Theta, \sigma}$ function also encapsulates measurement variables such as `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 $\mathbf{d}(\mathbf{e}_1 \dots \mathbf{e}_n)$, 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 \mathbf{d} (denoted by $\mathit{body}(\mathbf{d})$) substituting the formal parameters of the function (denoted by $\mathit{args}(\mathbf{d})$) with the values obtained evaluating $\mathbf{e}_1, \dots, \mathbf{e}_n$.

Rule [E-REP] implements internal state evolution through computational rounds: expression $\mathbf{rep}(\mathbf{e}_1)\{\mathbf{x} \Rightarrow \mathbf{e}_2\}$ evaluates to $\mathbf{e}_2[\mathbf{x} := \mathbf{v}]$ where \mathbf{v} is obtained from \mathbf{e}_1 on the first evaluation, and from the previous value of the whole \mathbf{rep} -expression on other evaluations.

Example 2.15. To illustrate rule [E-REP], as well as computational rounds, we consider program $\mathbf{rep}(1)\{\mathbf{x} \Rightarrow \ast(\mathbf{x}, 2)\}$. The first firing of a device δ is performed against the empty tree environment. Therefore, according to rule [E-REP], to evaluate $\mathbf{rep}(1)\{\mathbf{x} \Rightarrow \ast(\mathbf{x}, 2)\}$ means to evaluate the subexpression $\ast(1, 2)$, obtained from $\ast(\mathbf{x}, 2)$ by replacing \mathbf{x} with 1. This produces the value-tree $\theta = 2\langle 1, 2\langle 1, 2 \rangle \rangle$, where root 2 is the overall result as usual, while its sub-trees 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 $\mathbf{rep}(1)\{\mathbf{x} \Rightarrow \ast(\mathbf{x}, 2)\}$ at the second firing means evaluating the subexpression $\ast(2, 2)$, obtained from $\ast(\mathbf{x}, 2)$ by replacing \mathbf{x} with 2, which is the root of θ . Hence the results of computation are 2, 4, 8, and so on.

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

Example 2.16. To illustrate rule [E-NBR], consider $\mathbf{e}' = \mathbf{minHood}(\mathbf{nbr}\{\mathbf{snsNum}()\})$, where the 1-ary built-in function $\mathbf{minHood}$ returns the lower limit of values in the range of its neighbouring value argument, and the 0-ary built-in function \mathbf{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 mutually connected, while δ_A and δ_C are not connected;
- \mathbf{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 $\mathbf{snsNum}()$ on δ_A yields 1 (by rules [E-LOC] and [E-B-APP], since $(\mathbf{snsNum})_{\delta_A}^{\emptyset, \sigma}() = 1$); the evaluation of $\mathbf{nbr}\{\mathbf{snsNum}()\}$ on δ_A yields $(\delta_A \mapsto 1)\langle 1 \rangle$ (by rule [E-NBR], since no device has yet communicated with δ_A); and the evaluation of \mathbf{e}' on δ_A yields

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

(by rule [E-B-APP], since $(\mathbf{minHood})_{\delta_A}^{\emptyset, \sigma}(\delta_A \mapsto 1) = 1$). Therefore, at its first firing, 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

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

Suppose that device δ_B is the third device that fires. Then the evaluation of \mathbf{e}' on δ_B is performed with respect to the environment $\Theta_B = (\delta_A \mapsto \theta_A, \delta_C \mapsto \theta_C)$ and the evaluation of its subexpressions $\mathbf{nbr}\{\mathbf{snsNum}()\}$ and $\mathbf{snsNum}()$ 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 $(\mathbf{snsNum})_{\delta_B}^{\Theta''_B, \sigma}() = 2$; the evaluation of $\mathbf{nbr}\{\mathbf{snsNum}()\}$ 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 $(\mathbf{minHood})_{\delta_B}^{\Theta'_B, \sigma}(\phi) = 1$. Therefore the evaluation of \mathbf{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 firing of device δ_B will yield a value-tree with root 1 (which is the minimum of \mathbf{snsNum} across δ_A , δ_B and δ_C); any subsequent firing of device δ_A will yield a value-tree with root 1 (which is the minimum of \mathbf{snsNum} across δ_A and δ_B); and any subsequent firing of device δ_C will yield a value-tree with root 2 (which is the minimum of \mathbf{snsNum} across δ_B and δ_C).

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

3. NETWORK SEMANTICS

In [VAB⁺18], the overall network evolution was described in terms of an interleaving network semantics (INS for short). Unfortunately, the INS is not able to model every possible message interaction describable by an augmented event structure. Therefore, in this section we present a novel network semantics that overcomes this limitation. Namely, in Section 3.1 we present a true concurrent network semantics (TCNS for short) and then, in Section 3.2, we show that the TCNS is

- (1) a conservative extension of the INS given in [VAB⁺18], and
- (2) models every possible message interaction describable by an augmented event structure.

Because of (2) the TCNS is adequate for formalizing the relations between the **share** construct and the combined use of the **rep** and **nbr** constructs.

3.1. True Concurrent Network Semantics. The overall network evolution is formalized by the nondeterministic small-step operational semantics given in Figure 5 as a transition system on network configurations N . Figure 5 (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 summarizing the current values held by devices. The *activation predicate* α specifies whether each device is currently activated. Then, *Stat* (a pair of status field and activation predicate) models overall device status. 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 maps denoted as σ). Then, *Env* (a couple of topology and sensor state) models the system's environment. Finally, a whole network configuration N is a couple of a status and environment.

We use the following notation for maps. Let $\bar{x} \mapsto y$ denote a map sending each element in the sequence \bar{x} to the same element y . Let $m_0[m_1]$ denote the map with domain $\mathbf{dom}(m_0) \cup \mathbf{dom}(m_1)$ coinciding with m_1 in the domain of m_1 and with m_0 otherwise. Let

System configurations and action labels:		
Ψ	$::= \bar{\delta} \mapsto \bar{\Theta}$	status field
α	$::= \bar{\delta} \mapsto \bar{a}$ with $a \in \{\mathbf{false}, \mathbf{true}\}$	activation predicate
$Stat$	$::= \Psi, \alpha$	status
τ	$::= \bar{\delta} \mapsto \bar{I}$	topology
Σ	$::= \bar{\delta} \mapsto \bar{\sigma}$	sensors-map
Env	$::= \tau, \Sigma$	environment
N	$::= \langle Env; Stat \rangle$	network configuration
act	$::= \delta+ \mid \delta- \mid env$	action label
Environment well-formedness:		
$WFE(\tau, \Sigma)$ holds iff $\mathbf{dom}(\tau) = \mathbf{dom}(\Sigma)$ and $\tau(\delta) \subseteq \mathbf{dom}(\Sigma)$ for all $\delta \in \mathbf{dom}(\Sigma)$.		
Transition rules for network evolution:		$N \xrightarrow{act} N$
$\frac{[\mathbf{N-COMP}] \quad \alpha(\delta) = \mathbf{false} \quad \Theta' = F(\Psi(\delta)) \quad \delta; \Theta'; \Sigma(\delta) \vdash \mathbf{e}_{\mathbf{main}} \Downarrow \theta \quad \Theta = \Theta'[\delta \mapsto \theta]}{\langle \tau, \Sigma; \Psi, \alpha \rangle \xrightarrow{\delta+} \langle \tau, \Sigma; \Psi[\delta \mapsto \Theta], \alpha[\delta \mapsto \mathbf{true}] \rangle}$		
$\frac{[\mathbf{N-SEND}] \quad \alpha(\delta) = \mathbf{true} \quad \tau(\delta) = \bar{\delta} \quad \theta = \Psi(\delta)(\delta) \quad \Theta = \delta \mapsto \theta}{\langle \tau, \Sigma; \Psi, \alpha \rangle \xrightarrow{\delta-} \langle \tau, \Sigma; \Psi[\bar{\delta} \mapsto \Theta], \alpha[\delta \mapsto \mathbf{false}] \rangle}$		
$\frac{[\mathbf{N-ENV}] \quad WFE(Env') \quad Env' = \bar{\delta} \mapsto \bar{I}, \bar{\delta} \mapsto \bar{\sigma} \quad \Psi_0 = \bar{\delta} \mapsto \emptyset \quad \alpha_0 = \bar{\delta} \mapsto \mathbf{false}}{\langle Env; \Psi, \alpha \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi], \alpha_0[\alpha] \rangle}$		

Figure 5: Small-step operational true concurrent semantics for network evolution.

$m_0[m_1]$ (where m_i are maps to maps) denote the map with the *same domain* as m_0 made of $x \mapsto m_0(x)[m_1(x)]$ for all x in the domain of m_1 , $x \mapsto m_0(x)$ otherwise.

We consider transitions $N \xrightarrow{act} N'$ of three kinds: firing starts on a given device (for which act is $\delta+$ where δ is the corresponding device identifier), firing ends and messages are sent on a given device (for which act is $\delta-$), and environment changes, where act is the special label env . This is formalized in Figure 5 (bottom). Rule [N-COMP] (available for sleeping devices, i.e., with $\alpha(\delta) = \mathbf{false}$, and setting them to executing, i.e., $\alpha(\delta) = \mathbf{true}$) models a computation round at device δ : it takes the local value-tree environment filtered out of old values $\Theta' = F(\Psi(\delta))$; ⁵ then by the single device semantics it obtains the device's value-tree θ , which is used to update the system configuration of δ to $\Theta = \Theta'[\delta \mapsto \theta]$. It is worth observing that, although this rule updates a device's system configuration instantaneously, it models computations taking an arbitrarily long time, since the update is not visible until the following rule [N-SEND]. Notice also that all values used to compute θ are locally available (at the beginning of the computation), thus allowing for a fully-distributed implementation without global knowledge.

⁵Function $F(\Theta)$ in rule [N-FIR] models a filtering operation that clears out old stored values from the value-tree environment Θ , implicitly based on space/time tags. Notice that this mechanism allows messages to persist across rounds.

Remark 3.1 (On termination of device firing). 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 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). It is worth noticing that this assumption does not impact the results of this paper, since the programs that are relevant are terminating (a device performing a firing that does not terminate would be equivalent on a global network perspective to a shut-down device).

Rule [N-SEND] (available for running devices with $\alpha(\delta) = \mathbf{true}$, and setting them to non-running) models the message sending happening at the end of a computation round at a device δ . It takes the local value-tree $\theta = \Psi(\delta)(\delta)$ computed by last rule [N-COMP], and uses it to update neighbours' $\bar{\delta}$ values of $\Psi(\bar{\delta})$. Notice that the usage of α ensures that occurrences of rules [N-COMP] and [N-SEND] for a device are alternated.

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 5 (middle)—thus modelling node mobility as well as changes in environmental parameters. Let $\bar{\delta}$ be the domain of Env' . We first construct a status field Ψ_0 and an activation predicate α_0 associating to all the devices of Env' the empty context \emptyset and the **false** activation. Then, we adapt the existing status field Ψ and activation predicate α to the new set of devices: $\Psi_0[\Psi]$, $\alpha_0[\alpha]$ automatically handles removal of devices, mapping of new devices to the empty context and **false** activation, and retention of existing contexts and activation in the other devices. We remark that this rule is also used to model communication failure as topology changes.

Example 3.2. Consider a network of devices with $e' = \mathbf{minHood}(\mathbf{nbr}\{\mathbf{snsNum}()\})$ as introduced in Example 2.16. The network configuration illustrated at the beginning of Example 2.16 can be generated by applying rule [N-ENV] to the empty network configuration. I.e., we have

$$\langle \emptyset, \emptyset; \emptyset, \emptyset \rangle \xrightarrow{env} \langle Env_0; Stat_0 \rangle$$

where $Env_0 = \tau_0, \Sigma_0$, $Stat_0 = \Psi_0, \alpha_0$ and

- $\tau_0 = (\delta_A \mapsto \{\delta_B\}, \delta_B \mapsto \{\delta_A, \delta_C\}, \delta_C \mapsto \{\delta_B\})$,
- $\Sigma_0 = (\delta_A \mapsto (\mathbf{snsNum} \mapsto 1), \delta_B \mapsto (\mathbf{snsNum} \mapsto 2), \delta_C \mapsto (\mathbf{snsNum} \mapsto 3))$, and
- $\Psi_0 = (\delta_A \mapsto \emptyset, \delta_B \mapsto \emptyset, \delta_C \mapsto \emptyset)$,
- $\alpha_0 = (\delta_A \mapsto \mathbf{false}, \delta_B \mapsto \mathbf{false}, \delta_C \mapsto \mathbf{false})$.

Then, the three firings of devices δ_A , δ_C and δ_B illustrated in Example 2.16 correspond to the following transitions, respectively.

$$(1) \langle Env_0; \Psi_0, \alpha_0 \rangle \xrightarrow{\delta_A^+} \langle Env_0; \Psi_1, \alpha_A \rangle, \text{ where}$$

- $\theta_A = 1 \langle (\delta_A \mapsto 1) \langle 1 \rangle \rangle$;
- $\Psi_1 = (\delta_A \mapsto (\delta_A \mapsto \theta_A), \delta_B \mapsto \emptyset, \delta_C \mapsto \emptyset)$;
- $\alpha_A = (\delta_A \mapsto \mathbf{true}, \delta_B \mapsto \mathbf{false}, \delta_C \mapsto \mathbf{false})$.

$$(2) \langle Env_0; \Psi_1, \alpha_A \rangle \xrightarrow{\delta_A^-} \langle Env_0; \Psi_2, \alpha_0 \rangle, \text{ where}$$

- $\Psi_2 = (\delta_A \mapsto (\delta_A \mapsto \theta_A), \delta_B \mapsto (\delta_A \mapsto \theta_A), \delta_C \mapsto \emptyset)$.

$$(3) \langle Env_0; \Psi_2, \alpha_0 \rangle \xrightarrow{\delta_C^+} \langle Env_0; \Psi_3, \alpha_C \rangle, \text{ where}$$

- $\theta_C = 1 \langle (\delta_C \mapsto 3) \langle 3 \rangle \rangle$;
- $\Psi_3 = (\delta_A \mapsto (\delta_A \mapsto \theta_A), \delta_B \mapsto (\delta_A \mapsto \theta_A), \delta_C \mapsto (\delta_C \mapsto \theta_C))$;
- $\alpha_C = (\delta_A \mapsto \mathbf{false}, \delta_B \mapsto \mathbf{false}, \delta_C \mapsto \mathbf{true})$.

- (4) $\langle Env_0; \Psi_3, \alpha_C \rangle \xrightarrow{\delta_C^-} \langle Env_0; \Psi_4, \alpha_0 \rangle$, where
- $\Psi_4 = (\delta_A \mapsto (\delta_A \mapsto \theta_A), \delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_C \mapsto \theta_C), \delta_C \mapsto (\delta_C \mapsto \theta_C))$.
- (5) $\langle Env_0; \Psi_4, \alpha_0 \rangle \xrightarrow{\delta_B^+} \langle Env_0; \Psi_5, \alpha_B \rangle$, where
- $\theta_B = 1 \langle \phi \langle 2 \rangle \rangle$ where $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$;
 - $\Psi_5 = (\delta_A \mapsto (\delta_A \mapsto \theta_A), \delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C), \delta_C \mapsto (\delta_C \mapsto \theta_C))$;
 - $\alpha_B = (\delta_A \mapsto \mathbf{false}, \delta_B \mapsto \mathbf{true}, \delta_C \mapsto \mathbf{false})$.
- (6) $\langle Env_0; \Psi_5, \alpha_B \rangle \xrightarrow{\delta_B^-} \langle Env_0; \Psi_6, \alpha_0 \rangle$, where
- $\Psi_6 = (\delta_A \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B),$
 $\delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C),$
 $\delta_C \mapsto (\delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C))$,

Notice also that swapping the order of transitions δ_A^- and δ_C^+ would not change the following results, only their intermediate step Ψ'_2, α' where:

- $\Psi'_2 = (\delta_A \mapsto (\delta_A \mapsto \theta_A), \delta_B \mapsto \emptyset, \delta_C \mapsto (\delta_C \mapsto \theta_C))$;
- $\alpha' = (\delta_A \mapsto \mathbf{true}, \delta_B \mapsto \mathbf{false}, \delta_C \mapsto \mathbf{true})$.

3.2. Properties of the Network Semantics. The INS given in [VAB⁺18] can be modeled by replacing the rules [N-COMP] and [N-SEND] of the TCNS in Figure 5 by the following single rule [N-FIR] modelling an instantaneous round of computation (including both computing and sending messages):

$$\frac{[\text{N-FIR}] \quad \alpha(\delta) = \mathbf{false} \quad \tau(\delta) = \bar{\delta} \quad \Theta' = F(\Psi(\delta)) \quad \delta; \Theta'; \Sigma(\delta) \vdash \mathbf{e}_{\text{main}} \Downarrow \theta \quad \Theta = \delta \mapsto \theta}{\langle \tau, \Sigma; \Psi, \alpha \rangle \xrightarrow{\delta} \langle \tau, \Sigma; \Psi[\delta \mapsto \Theta'][\bar{\delta} \mapsto \Theta], \alpha \rangle}$$

and by considering only network statuses $\langle Env; \Psi, \alpha \rangle$ where $\alpha = \bar{\delta} \mapsto \mathbf{false}$.⁶ Notice that this restriction is consistent since rules [N-FIR] and [N-ENV] both preserve the condition $\alpha = \bar{\delta} \mapsto \mathbf{false}$.

The TCNS is a conservative extension of the INS, extending it to model non-instantaneous rounds of computations by splitting the *computation* and *sending* parts. This is formally stated by the following theorem.

Theorem 3.3 (TCNS is a conservative extension of INS). *Let $N = \langle Env; \Psi, \alpha \rangle$ be a TCNS network configuration such that $\alpha(\delta) = \mathbf{false}$. Then a sequence of δ^+ and δ^- transitions $N \xrightarrow{\delta^+} N' \xrightarrow{\delta^-} N''$ (rules [N-COMP], [N-SEND]) leads to the same configuration N'' as the single δ transition $N \xrightarrow{\delta} N''$ (rule [N-FIR]).*

Thus, any INS system evolution $N_1 \xrightarrow{\text{act}} \dots \xrightarrow{\text{act}} N_n$ corresponds to an analogous TCNS system evolution where each δ transition is replaced by a pair of δ^+ and δ^- transitions.

Proof. Assume that $Env = \tau, \Sigma$ and $\tau(\delta) = \bar{\delta}$. Furthermore, suppose that $\Theta' = F(\Psi(\delta))$, $\delta; \Theta'; \Sigma(\delta) \vdash \mathbf{e}_{\text{main}} \Downarrow \theta$, $\Theta = \delta \mapsto \theta$ and $\Theta'' = \Theta'[\Theta]$.

Then by rule [N-COMP], $N' = \langle Env; \Psi', \alpha' \rangle$ where $\Psi' = \Psi[\delta \mapsto \Theta'] = \Psi[\delta \mapsto \Theta'][\bar{\delta} \mapsto \Theta]$ and $\alpha' = \alpha[\delta \mapsto \mathbf{true}]$. Finally, by rule [N-SEND], $N'' = \langle Env; \Psi'', \alpha'' \rangle$ where:

- $\Psi'' = \Psi'[\bar{\delta} \mapsto \Theta] = \Psi[\delta \mapsto \Theta'][\bar{\delta} \mapsto \Theta][\bar{\delta} \mapsto \Theta] = \Psi[\delta \mapsto \Theta'][\bar{\delta} \mapsto \Theta]$
- $\alpha'' = \alpha'[\delta \mapsto \mathbf{false}] = \alpha[\delta \mapsto \mathbf{true}][\delta \mapsto \mathbf{false}] = \alpha$.

Thus, N'' is the same as in the conclusion of rule [N-FIR]. □

⁶Actually, in the INS rules given in [VAB⁺18] there is no activation predicate α .

Notice that every (TCNS or INS) system evolution implies an underlying augmented event structure (c.f. Definition 2.3) describing its message passing details, as per the following definition.

Definition 3.4 (Space-Time Value Underlying a System Evolution). Let $\mathcal{S} = N_0 \xrightarrow{act_1} \dots \xrightarrow{act_n} N_n$ with $N_0 = \langle \emptyset, \emptyset; \emptyset, \emptyset \rangle$ be any system evolution. We say that:

- $D = \{\delta \mid \exists i. act_i = \delta + \vee act_i = \delta -\}$ are the device identifiers appearing in \mathcal{S} ;
- $C^\delta = \langle i \leq n \mid act_i = \delta + \rangle$ are the indexes of transitions applying rule [N-COMP];
- $S^\delta = \langle i \leq n \mid act_i = \delta - \rangle$ are the indexes of transitions applying rule [N-SEND];
- $E = \{\langle \delta, i \rangle \mid \delta \in D \wedge 1 \leq i \leq |C^\delta|\}$ is the set of events in \mathcal{S} ;
- $d : E \rightarrow D$ maps each event $\epsilon = \langle \delta, i \rangle$ to the device δ where it is happening;
- $\epsilon_1 \rightsquigarrow \epsilon_2$ where $\epsilon_k = \langle \delta_k, i_k \rangle$ and $j_1 = S_{i_1}^{\delta_1}, j_2 = C_{i_2}^{\delta_2}$ if and only if:
 - N_{j_1} has topology τ such that $\delta_2 \in \tau(\delta_1)$ (the message from ϵ_1 reaches δ_2),
 - there is no $j' \in (j_1; j_2]$ with $j' \in S^{\delta_1}$ and $N_{j'}$ with topology τ such that $\delta_2 \in \tau(\delta_1)$ (there are no more recent messages from δ_1 to ϵ_2),
 - for every $j' \in (j_1; j_2]$ with $j' \in S^{\delta_2}$ and $N_{j'}$ with status field Ψ , then $\delta_1 \in \mathbf{dom}(\Psi(\delta_2))$ (the message from ϵ_1 to δ_2 is not filtered out as obsolete before ϵ_2);
- $<$ is the transitive closure of \rightsquigarrow ;
- $f : E \rightarrow \mathbf{V}$ is such that $f(\langle \delta, i \rangle) = \rho(\Psi(\delta)(\delta))$ where $N_{C_i^\delta} = \langle Env, \Psi, \alpha \rangle$.

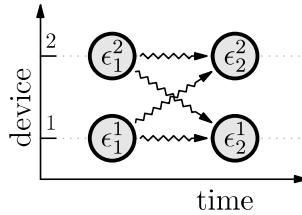
Then we say that \mathcal{S} follows $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$, and $\Phi = \langle \mathbf{E}, f \rangle$ is the *space-time value underlying to \mathcal{S}* .

Notice that the \mathbf{E} and Φ defined above are unique given \mathcal{S} . Furthermore, as stated by the following theorem, the TCNS is sufficiently expressive to model every possible message interaction describable by an augmented event structure.

Theorem 3.5 (TCNS completeness). *Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be an augmented event structure. Then there exist (infinitely many) system evolutions following \mathbf{E} .*

Proof. See Appendix A. □

Notice as well that this expressiveness is not the case for INS. For example, no INS system evolution can follow this augmented event structure:



In fact, the transitions corresponding to ϵ_1^1 would need to have $\tau(\delta_i) = \{\delta_1, \delta_2\}$, since both events reach both devices. Then if w.l.o.g. the transition corresponding to ϵ_1^1 happens before the one corresponding ϵ_1^2 , since $\epsilon_1^1 \rightsquigarrow \epsilon_1^2$ does not hold, the transition corresponding to ϵ_1^2 must filter out the message coming from δ_1 . It follows that ϵ_1^1 does not reach ϵ_2^2 as well, a contradiction.

4. THE SHARE CONSTRUCT

Section 4.1 explains and illustrates the problematic interaction between time evolution and neighbour interaction constructs. Section 4.2 then shows how the `share` construct overcomes this problematic interaction. Section 4.3 presents the operational semantics of the `share` construct. Section 4.4 introduces automatic rewritings of `rep` constructs into `share` constructs: two preserving the behavior, thus showing that `share` has the expressive power to substitute most usages of `rep` and `nbr` in programs; and one changing the behavior (in fact, improving it in many cases). Section 4.5 demonstrates the automatic behavior improvement for the example in Section 4.1, while estimating the general communication speed improvement induced by the rewriting. Section 4.6 shows examples for which the rewriting fails to preserve the intended behavior, and Section 4.7 concludes by showing that behavior is preserved for the relevant subset of field calculus pinpointed in [VAB⁺18].

4.1. Problematic Interaction between `rep` and `nbr` Constructs. Unfortunately, the apparently straight-forward combination of state evolution with `nbr` and state sharing with `rep` turns out to contain a hidden delay, which was identified and explained in [ABDV18]. This problem may be illustrated by attempting to construct a simple function that spreads information from an event as quickly as possible. Let us say there is a Boolean space-time value `condition`, and we wish to compute a space-time function `ever` that returns true precisely at events where `condition` is true and in the causal future of those events—i.e., spreading out at the maximum theoretical speed throughout the network of devices. One might expect this could be implemented as follows in field calculus:

```
def ever1(condition) {
  rep (false) { (old) => anyHoodPlusSelf(nbr{old}) || condition }
}
```

where `anyHoodPlusSelf` is a built-in function that returns true if any value is true in its neighbouring value input (including the value `old` held for the current device). Walking through the evaluation of this function, however, reveals that there is a hidden delay. In each round, the `old` variable is updated, and will become true if either `condition` is true now for the current device or if `old` was true in the previous round for the current device or for any of its neighbours. Once `old` becomes true, it stays true for the rest of the computation. Notice, however, that a neighbouring device does not actually learn that `condition` is true, but that `old` is true. In an event where `condition` first becomes true, the value of `old` that is shared is still false, since the `rep` does not update its value until after the `nbr` has already been evaluated. Only in the next round do neighbours see an updated value of `old`, meaning that `ever1` is not spreading information fast enough to be a correct implementation of `ever`.

We might try to improve this routine by directly sharing the value of `condition`:

```
def ever2(condition) {
  rep (false) { (old) => anyHoodPlusSelf(nbr{old || condition}) }
}
```

This solves the problem for immediate neighbours, but does not solve the problem for neighbours of neighbours, which still have to wait an additional round before `old` is updated (see Example 4.1 for a sample execution of these functions, showcasing how some devices realise that `condition` has become true with a delay).

In fact, in order to avoid delays, communication cannot use `rep` but only `nbr`. Since a single `nbr` can only reach values in immediate neighbours, in order to reach values in the arbitrary past of a device, it is necessary to use an arbitrary number of nested `nbr` statements (each of them contributing to the total message size exchanged). This can be achieved by using unbounded recursion, as previously outlined in [ABDV18]:

```
def ever3(condition) {
  let new = anyHoodPlusSelf(nbr{condition}) in
  if (countHood() == 0) { new } { ever3(new) }
}
```

where `countHood` counts the number of neighbours, i.e., determining whether any neighbour has reached the same depth of recursion in the branch. Thus, in `ever3`, neighbours' values of `condition` are fed to a nested call to `ever3` (if there are any); and this process is iterated until no more values to be considered are present. This function therefore has a recursion depth equal to the longest sequence of events $\epsilon_0 \rightsquigarrow \dots \rightsquigarrow \epsilon$ ending in the current event ϵ , inducing a linearly increasing computational time and message size and making the routine effectively infeasible for long-running systems.

This case study illustrates the more general problem of delays induced by the interaction of `rep` and `nbr` constructs in field calculus, as identified in [ABDV18]. With these constructs, it is never possible to build computations involving long-range communication that are as fast as possible and also lightweight in the amount of communication required.

4.2. Beyond `rep` and `nbr`. In order to overcome the problematic interaction between `rep` and `nbr`, we propose a new construct that combines aspects of both:

$$\mathbf{share}(e_1)\{x \Rightarrow e_2\}$$

where: e_1 is the initial local expression; x is the state variable, holding a neighbouring value; e_2 is an aggregation expression, taking x and producing a local value; and the whole expected result is a local value. Informally, at each firing, `share` works in the following way:

- (1) it constructs a neighbouring value ϕ with the outcomes of its evaluation in neighbouring events (cf. Def. 2.1)—namely, ϕ maps the local device to the result of this `share` at the previous round (or, if absent, to e_1 as with `rep`), and the neighbouring devices to the latest available result of this `share` (involving communication of values as with `nbr`); and
- (2) it evaluates the aggregation expression e_2 by using ϕ as the value of x to obtain a local result, which is both sent to neighbours (for their future rounds) and kept locally (for the next local firing).

So, although the syntactic structure of the `share` construct is identical to that of `rep`, the two constructs differ in the way the construct variable x is interpreted at each firing:

- in `rep`, the value of x is the local value produced by evaluating the construct in the previous round, or the result of evaluating e_1 if there is no prior-round value;
- in `share`, instead, x is a *neighbouring value* comprising that same value for the current device plus the values of the construct produced by neighbours in their most recent evaluation (thus `share` incorporates communication as well).

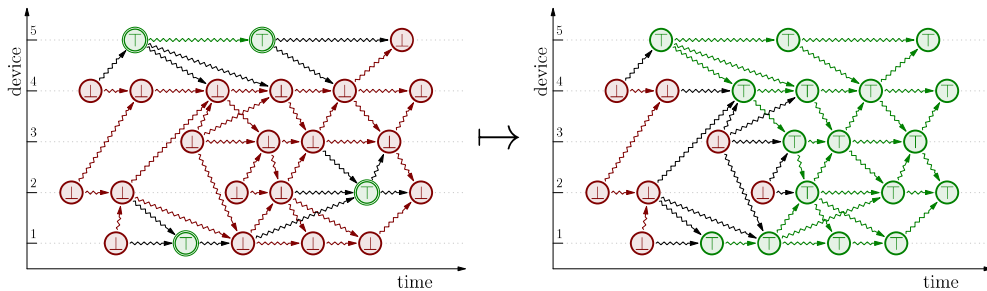
Moreover, in `share`, e_2 is responsible for *aggregating* the neighbouring value x into a local value that is shared with neighbours at the end of the evaluation (thus `share` incorporates aggregation as well).

As illustrated by the following example, using the `share` construct allows to overcome the problematic interaction between `rep` and `nbr` (see Section 4.1).

Example 4.1 (Share Construct). Consider the body of function `ever`:

```
def ever(condition) {
  share (false) { (old) => anyHoodPlusSelf(old) || condition }
}
```

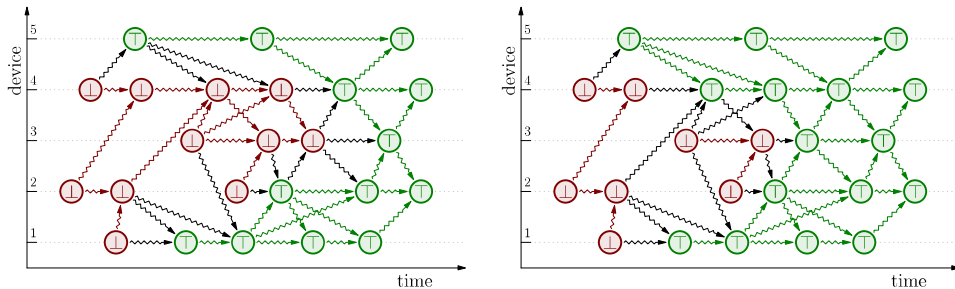
Assume this program is run on a network of 5 devices, executing rounds according to the following augmented event structure (condition input values are on the left, output of the `ever` function is on the right):



At the first round of any device δ , no messages has been received yet, thus the `share` construct is evaluated by substituting `old` with the neighbouring value $\delta \mapsto \perp$. It follows that `anyHoodPlusSelf(old)` is false, hence the result of the whole construct is equal to `condition` (which is true only for $\delta = 5$). After the computation is complete, the result of the `share` construct is sent to neighbours.

At the second round of device 4, the only message received is a `false` from device 2 (and another `false` persisting from device 4 itself), thus the overall result is still false. At the third round of device 4, a `true` message from device 5 joins the pool, switching the overall result to `true`. In following rounds, there is always a `true` message persisting from device 4 itself, so the result stays true. Similar reasoning can be applied to the other devices.

Notice that the outputs of the `ever1` (left) and `ever2` (right) functions, from Section 4.1, would instead be:



In `ever1`, devices 3 and 4 converge to \top with two rounds of delay; while in `ever2` device 3 converges to \top with one round of delay. Function `ever3`, instead, behaves exactly as `ever`, although requiring unbounded recursion depth (hence greater computational complexity in every round).

Auxiliary functions:		
$\phi_0[\phi_1] = \phi_2$ where $\phi_2(\delta) = \begin{cases} \phi_1(\delta) & \text{if } \delta \in \mathbf{dom}(\phi_1) \\ \phi_0(\delta) & \text{otherwise} \end{cases}$		
Rule for expression evaluation:		
[E-SHARE]	$\delta; \pi_1(\Theta); \sigma \vdash \mathbf{e}_1 \Downarrow \theta_1$ $\delta; \pi_2(\Theta); \sigma \vdash \mathbf{e}_2[x := \phi] \Downarrow \theta_2$	$\phi' = \rho(\pi_2(\Theta))$ $\phi = (\delta \mapsto \rho(\theta_1))[\phi']$
$\delta; \Theta; \sigma \vdash \mathbf{share}(\mathbf{e}_1)\{\mathbf{x} \Rightarrow \mathbf{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle$		

Figure 6: Operational semantics for the **share** construct.

4.3. Operational Semantics. Formal operational semantics for the **share** construct is presented in Figure 6 (bottom frame), as an extension to the semantics given in Section 2.4. The evaluation rule is based on the auxiliary functions given in Figure 6 (top frame), plus the auxiliary functions in Figure 4 (second frame). In particular, we use the notation $\phi_0[\phi_1]$ to represent “field update”, so that its result ϕ_2 has $\mathbf{dom}(\phi_2) = \mathbf{dom}(\phi_0) \cup \mathbf{dom}(\phi_1)$ and coincides with ϕ_1 on its domain, or with ϕ_0 otherwise.

The evaluation rule [E-SHARE] produces a value-tree with two branches (for \mathbf{e}_1 and \mathbf{e}_2 respectively). First, it evaluates \mathbf{e}_1 with respect to the corresponding branches of neighbours $\pi_1(\Theta)$ obtaining θ_1 . Then, it collects the results for the construct from neighbours into the neighbouring value $\phi' = \rho(\pi_2(\Theta))$. In case ϕ' does not have an entry for δ , $\rho(\theta_1)$ is used obtaining $\phi = (\delta \mapsto \rho(\theta_1))[\phi']$. Finally, ϕ is substituted for \mathbf{x} in the evaluation of \mathbf{e}_2 (with respect to the corresponding branches of neighbours $\pi_2(\Theta)$) obtaining θ_2 , setting $\rho(\theta_2)$ to be the overall value.

Example 4.2 (Operational Semantics). Consider the body of function **ever**:

```
def ever(condition) {
  share (false) { (old) => anyHoodPlusSelf(old) || condition }
}
```

Suppose that device $\delta = 0$ first executes a round of computation without neighbours (i.e., Θ is empty), and with **condition** equal to **false**. The evaluation of the **share** construct proceeds by evaluating **false** into $\theta_1 = \mathbf{false}\langle\rangle$, gathering neighbour values into $\phi' = \bullet$ (no values are present), and adding the value for the current device obtaining $\phi = (0 \mapsto \mathbf{false})[\bullet] = 0 \mapsto \mathbf{false}$. Finally, the evaluation completes by storing in θ_2 the result of **anyHoodPlusSelf**($0 \mapsto \mathbf{false}$)||**false** (which is **false** $\langle\dots\rangle$ ⁷). At the end of the round, device 0 sends a broadcast message containing the result of its overall evaluation, and thus including $\theta^0 = \mathbf{false}\langle\mathbf{false}, \mathbf{false}\langle\dots\rangle\rangle$.

Suppose now that device $\delta = 1$ receives the broadcast message and then executes a round of computation where **condition** is **true**. The evaluation of the **share** constructs starts similarly as before with $\theta_1 = \mathbf{false}\langle\rangle$, $\phi' = 0 \mapsto \mathbf{false}$, $\phi = 0 \mapsto \mathbf{false}, 1 \mapsto \mathbf{false}$. Then the body of the **share** is evaluated as **anyHoodPlusSelf**($0 \mapsto \mathbf{false}, 1 \mapsto \mathbf{false}$)||**true** into θ_2 , which is **true** $\langle\dots\rangle$. At the end of the round, device 1 broadcasts the result of its overall evaluation, including $\theta^1 = \mathbf{true}\langle\mathbf{false}, \mathbf{true}\langle\dots\rangle\rangle$.

⁷We omit the part of the value tree that are produced by semantic rules not included in this paper, and refer to[VAB⁺18, Electronic Appendix] for the missing parts.

Then, suppose that device $\delta = 0$ receives the broadcast from device 1 and then performs another round of computation with `condition` equal to `false`. As before, $\theta_1 = \text{false}\langle \rangle$, $\phi = \phi' = 0 \mapsto \text{false}, 1 \mapsto \text{true}$ and the body is evaluated as `anyHoodPlusSelf(0 \mapsto false, 1 \mapsto true) || false` which produces `true` $\langle \dots \rangle$ for an overall result of $\theta^2 = \text{true}\langle \text{false}, \text{true}\langle \dots \rangle \rangle$.

Finally, suppose that device $\delta = 1$ does not receive that broadcast and discards 0 from its list of neighbours before performing another round of computation with `condition` equal to `false`. Then, $\theta_1 = \text{false}\langle \rangle$, $\phi' = 1 \mapsto \text{true}$, $\phi = (1 \mapsto \text{false})[1 \mapsto \text{true}] = 1 \mapsto \text{true}$, and the body is evaluated as `anyHoodPlusSelf(1 \mapsto true) || false` which produces `true` $\langle \dots \rangle$.

4.4. Automatic Rewritings of rep Constructs into share Constructs. The `share` construct can be automatically incorporated into programs using `rep` and `nbr` in a few ways. First, we may want to rewrite a program while maintaining the behavior unchanged, thus showing that the expressive power of `share` is enough to replace other constructs to some extent. In particular, we can fully replace the `rep` construct through the following rewriting, expressed through the notation $e[e_1 := e'_1, \dots, e_n := e'_n]$ representing an expression e in which the distinct subexpressions e_1, \dots, e_n have been simultaneously replaced by the corresponding expressions e'_1, \dots, e'_n —if e_i is a subexpression of e_j (for some $i \neq j$) then the occurrences e_j are replaced by e'_j .

Rewriting 4.3 (`rep`-elimination).

$$\text{rep}(e_1)\{(x) \Rightarrow e_2\} \longrightarrow \text{share}(e_1)\{(x) \Rightarrow e_2[x := \text{localHood}(x)]\}$$

where `localHood` is a built-in operator that given a neighbouring value ϕ returns the local value $\phi(\delta)$ for the current device.

Theorem 4.4. *Rewriting 4.3 preserves the program behavior.*

Proof. Correctness follows since the value $\phi(\delta)$ in the neighbouring value ϕ substituted for x in the `share` construct corresponds exactly to the value that is substituted for x in the corresponding `rep` construct. \square

In addition to eliminating `rep` occurrences, the `share` construct is able to factor out many common usages of the `nbr` construct as well (even though not all of them), as per the following equivalent rewriting. For ease of presentation, we extend the syntax of `share` to handling multiple input-output values: `share(e1, e2)\{(x1, x2) \Rightarrow e'1, e'2\}`, to be interpreted as a shorthand for a single-argument construct where the multiple input-output values have been gathered into a tuple (unpacking them before computing e'_1, e'_2 and then packing their result).

Rewriting 4.5 (`nbr`-elimination).

$$\begin{aligned} \text{rep}(e_1)\{(x) \Rightarrow e_2\} &\longrightarrow \\ \text{fst}(\text{share}(e_1, e_1)\{(x, y) \Rightarrow & \\ \quad e_2[x := \text{localHood}(x), \text{nbr}\{x\} := \text{localChange}(y, \text{localHood}(x))], & \\ \quad \text{localHood}(x) & \\ \quad \}) & \end{aligned}$$

where y is a fresh variable and `localChange`(ϕ, ℓ) updates the value of ϕ for the current device δ with ℓ , returning $\phi[\delta \mapsto \ell]$.

Theorem 4.6. *Rewriting 4.5 preserves the program behavior.*

Proof. We prove by induction that the two components of the `share` translation correspond to the `rep` current and previous results (respectively, using e_1 if no such previous value is available). On initial rounds of evaluation, the `share` construct evaluates to $e_2[x := e_1, \text{nbr}\{x\} := \text{nbr}\{e_1\}], e_1$ (by substituting x, y by e_1), as the `rep` construct. On other rounds, the second component of `share` is `localHood(x)`, which is the previous result of the first component of `share`, which is the previous result of the `rep` construct by inductive hypothesis. Furthermore, the first component of `share` is e_2 with arguments `localHood(x)` (again, the previous result of the `rep` construct) and `localChange(y, localHood(x))`, which is the neighbours' values for the second argument together with the previous value of the `rep` construct for the current device. On the other hand, `nbr{x}` is the neighbours' values for the old value of the `rep` construct, together with the local previous value of the `rep` construct. By inductive hypothesis, the two things coincide, concluding the proof. \square

However, a more interesting rewriting is the following *non-equivalent* one, which for many algorithms is able to automatically improve the communication speed while preserving the overall meaning.

Rewriting 4.7 (non-equivalent).

$$\text{rep}(e_1)\{(x) \Rightarrow e_2\} \longrightarrow \text{share}(e_1)\{(x) \Rightarrow e_2[x := \text{localHood}(x), \text{nbr}\{x\} := x]\}$$

In particular, we shall see in Section 4.5 how this rewriting translates the inefficient `ever1` routine into a program equivalent to `ever3`, and in Section 4.7 that this rewriting preserves the eventual behavior of a whole fragment of field calculus programs, while improving its efficiency. In particular, the improvement in communication speed can be estimated to be at least three-fold (see Section 4.5). Unfortunately, programs may exist for which this translation fails to preserve the intended meaning (see Section 4.6). This usually happens for time-based algorithms where the one-round delay is incorporated into the logic of the algorithm, or weakly characterised functions which may need reduced responsiveness for allowing results to stabilise. Thus, better performing alternatives using `share` may still exist after the program logic has been accordingly fixed.

4.5. The `share` Construct Improves Communication Speed. To illustrate how `share` solves the problem illustrated in Section 4.1, let us once again consider the `ever` function discussed in that section, for propagating when a `condition` Boolean has ever become true. By applying Rewriting 4.7 to the `ever1` function introduced in Section 4.1 we obtain exactly the `ever` function introduced in Section 4.3:

```
def ever(condition) {
  share (false) { (old) => anyHoodPlusSelf(old) || condition }
}
```

Function `ever` is simultaneously (i) compact and readable, even more so than `ever1` and `ever2` (note that we no longer need to include the `nbr` construct); (ii) lightweight, as it involves the communication of a single Boolean value each round and few operations; and (iii) optimally efficient in communication speed, since it is true for any event ϵ with a causal predecessor $\epsilon' \leq \epsilon$ where `condition` was true. In particular

- in such an event ϵ' the overall `share` construct is true, since it goes to

```
anyHoodPlusSelf(old) || true
```

regardless of the values in `old`;

- in any subsequent event ϵ'' (i.e. $\epsilon' \rightsquigarrow \epsilon''$) the `share` construct is true since the field value `old` contains a true value (the one coming from ϵ'), and
- the same holds for further following events ϵ by inductive arguments.

In field calculus without `share`, such optimal communication speed can be achieved only through unbounded recursion, as argued in [ABDV18] and reviewed above in Section 4.1.

As a further example of successful application of Rewriting 4.7, consider the following routine where `maxHoodPlusSelf` is a built-in function returning the maximum value in the range of a numeric neighbouring value.

```
def sharedcounter1() {
  rep (0) { (old) => max(maxHoodPlusSelf(nbr{old}), rep(0){(c)=>c+1}) }
}
```

This function computes a local counter through `rep(0)\{(c)=>c+1\}` and then uses it to compute the maximum number of rounds a device in the network has performed (even though information about the number of rounds for other devices propagates at reduced speed). If we rewrite this function by eliminating the first `rep` through Rewriting 4.7, we obtain:

```
def sharedcounter2() {
  share (0) { (old) => max(maxHoodPlusSelf(old), rep(0){(c)=>c+1}) }
}
```

where information about the number of rounds for other devices is propagated to neighbours at the full multi-path speed allowed by `share`. It is worth observing that eliminating the remaining `rep` by further applying Rewriting 4.7 would produce the same result of applying Rewriting 1, i.e.:

```
def sharedcounter() {
  share (0) { (old) => max(maxHoodPlusSelf(old), share(0){(c)=>localHood(c)+1}) }
}
```

and therefore would not affect the information propagation speed.

The average improvement in communication speed of a routine being converted from the usage of `rep + nbr` to `share` according to Rewriting 4.7 can also be statistically estimated, depending on the communication pattern used by the routine.

An algorithm follows a *single-path* communication pattern if its outcome in an event depends essentially on the value of a single selected neighbour: prototypical examples of such algorithms are distance estimations [ADV17, ADV18, ACDV17], which are computed out of the value of the single neighbour on the optimal path to the source. In this case, letting T be the average interval between subsequent rounds, the expected communication delay of an hop is $T/2$ with `share` (since it can randomly vary from 0 to T) and $T/2 + T = 3/2T$ with `rep + nbr` (since a full additional round T is wasted in this case). Thus, the usage of `share` allows for an expected three-fold improvement in communication speed for these algorithms.

An algorithm follows a *multi-path* communication pattern if its outcome in an event is obtained from the values of all neighbours: prototypical examples of such algorithms are data collections [ABDV19], especially when they are idempotent (e.g. minimums or maximums). In this case, the existence of a single communication path $\epsilon_0 \rightsquigarrow \dots \rightsquigarrow \epsilon$ is sufficient for the

value in ϵ_0 to be taken into account in ϵ . Even though the delay of any one of such paths follows the same distribution as for single-path algorithms (0 to T per step with `share`, T to $2T$ per step with `rep + nbr`), the overall delay is *minimized* among each existing path. It follows that for sufficiently large numbers of paths, the delay is closer to the minimum of a single hop (0 with `share`, T with `rep + nbr`) resulting in an even larger improvement.

4.6. Limitations of the Automatic Rewriting. In the previous section, we showed how the non-equivalent rewriting of `rep + nbr` statements into `share` statements is able to improve the performance of algorithms, both in the specific case of the `ever` and `sharedcounter` functions, and statistically for the communication speed of general algorithms. However, this procedure may *not* work for all functions: for example, consider the following routine

```
def fragilesharedcounter() {
  rep (0) { (old) => maxHoodPlusSelf(nbr{old})+1 }
}
```

that, if the scheduling of computation rounds is sufficiently regular across the network, is able to approximate the maximum number of rounds a device in the network has performed (even though information about the number of rounds for other devices propagates at reduced speed). If we rewrite this function through Rewriting 4.7, we obtain:

```
def fragilesharedcounter1() {
  share (0) { (old) => maxHoodPlusSelf(old)+1 }
}
```

which does *not* approximate the same quantity. Instead, it computes the maximum length of a path of messages reaching the current event, which may be unboundedly higher than round counts in case of dense networks.

In fact, the fragile shared counter function is a paradigmatic example of rewriting failure: it is a time-based function, whose results are strongly altered by removing the one-round wait generated by `rep + nbr`. Another class of programs for which the rewriting fails is that of functions with weakly defined behavior, usually based on heuristics, for which the increase in responsiveness may increase the fluctuations in results (or even prevent stabilisation to a meaningful value).

4.7. The share Construct Preserves Self-stabilisation. In this section, we prove that the automatic rewriting is able to improve an important class of functions with strongly defined behavior: the *self-stabilising fragment* of field calculus identified in [VAB⁺18]. Functions complying to the syntactic and semantic restrictions imposed by this fragment are guaranteed to be *self-stabilising*, that is, whenever the function inputs and network structure stop changing, the output values will eventually converge to a value which only depends on the limit inputs and network structure (and not on what happened before the convergence of the network). This property captures the ability of a function to react to input changes, self-adjusting to the new correct value, and is thus a commonly used notion for strongly defining the behavior of a distributed function.

Definition 2.10 formalises the notion of self-stabilisation for space-time functions. This definition can be translated to field calculus functions and expressions by means of Definition 3.4, as in the following definition:

$ \begin{array}{l} \mathbf{s} ::= \mathbf{x} \mid \mathbf{v} \mid \text{let } \mathbf{x} = \mathbf{s} \text{ in } \mathbf{s} \mid \mathbf{f}(\bar{\mathbf{s}}) \mid \text{if}(\mathbf{s})\{\mathbf{s}\}\{\mathbf{s}\} \mid \text{nbr}\{\mathbf{s}\} \\ \mid \text{rep}(\mathbf{e})\{\mathbf{x} \Rightarrow \mathbf{f}^{\mathbf{C}}(\text{nbr}\{\mathbf{x}\}, \text{nbr}\{\mathbf{s}\}, \bar{\mathbf{e}})\} \\ \mid \text{rep}(\mathbf{e})\{\mathbf{x} \Rightarrow \mathbf{f}(\text{mux}(\text{nbr}\{\mathbf{l}\}(\mathbf{s}), \text{nbr}\{\mathbf{x}\}, \mathbf{s}), \bar{\mathbf{s}})\} \\ \mid \text{rep}(\mathbf{e})\{\mathbf{x} \Rightarrow \mathbf{f}^{\mathbf{R}}(\text{minHoodLoc}(\mathbf{f}^{\mathbf{MP}}(\text{nbr}\{\mathbf{x}\}, \bar{\mathbf{s}}), \mathbf{s}), \mathbf{x}, \bar{\mathbf{e}})\} \end{array} $	self-stab. expr. with rep
$ \begin{array}{l} \mathbf{s} ::= \mathbf{x} \mid \mathbf{v} \mid \text{let } \mathbf{x} = \mathbf{s} \text{ in } \mathbf{s} \mid \mathbf{f}(\bar{\mathbf{s}}) \mid \text{if}(\mathbf{s})\{\mathbf{s}\}\{\mathbf{s}\} \mid \text{nbr}\{\mathbf{s}\} \\ \mid \text{share}(\mathbf{e})\{\mathbf{x} \Rightarrow \mathbf{f}^{\mathbf{C}}(\mathbf{x}, \text{nbr}\{\mathbf{s}\}, \bar{\mathbf{e}})\} \\ \mid \text{share}(\mathbf{e})\{\mathbf{x} \Rightarrow \mathbf{f}(\text{mux}(\text{nbr}\{\mathbf{l}\}(\mathbf{s}), \mathbf{x}, \mathbf{s}), \bar{\mathbf{s}})\} \\ \mid \text{share}(\mathbf{e})\{\mathbf{x} \Rightarrow \mathbf{f}^{\mathbf{R}}(\text{minHoodLoc}(\mathbf{f}^{\mathbf{MP}}(\mathbf{x}, \bar{\mathbf{s}}), \mathbf{s}), \text{localHood}(\mathbf{x}), \bar{\mathbf{e}})\} \end{array} $	self-stab. expr. with share

Figure 7: Syntax of the self-stabilising fragment of field calculus introduced in [VAB⁺18], together with its translation through Rewriting 4.7. Self-stabilising expressions \mathbf{s} occurring inside **rep** and **share** statements cannot contain free occurrences of the **share**-bound variable \mathbf{x} .

Definition 4.8 (Stabilising Expression). A field calculus expression \mathbf{e} is *stabilising* with limit Ψ on \mathbf{G} iff for any system evolution \mathcal{S} of program \mathbf{e} following \mathbf{E} with limit \mathbf{G} , the space-time value Φ corresponding to \mathcal{S} is stabilising with limit Ψ . Similarly, a field calculus function $\mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n)$ is *self-stabilising* with limit $\mathbf{g} : \mathbf{V}(\mathbf{G})^n \rightarrow \mathbf{V}(\mathbf{G})$ iff given any stabilising $\langle \mathbf{e}_1, \dots, \mathbf{e}_n \rangle$ with limit $\langle \Psi_1, \dots, \Psi_n \rangle$, $\mathbf{f}(\mathbf{e}_1, \dots, \mathbf{e}_n)$ is stabilising with limit $\Psi = \mathbf{g}(\Psi_1, \dots, \Psi_n)$.

For example, function **ever** is not self-stabilising: if the inputs stabilise to being false everywhere, the function output could still be true if some past input was indeed true. As a positive example, the following function is self-stabilising, and computes the hop-count distance from the closest device where **source** is true.

```

def hopcount(source) {
  share (infinity) { (old) => mux(source, 0, minHood(old)+1) }
}

```

Here, **minHood** computes the minimum in the range of a numeric neighbouring value (excluding the current device), while **mux** (multiplexer) selects between its second and third argument according to the value of the first (similarly as **if**, but evaluating all arguments).

A rewriting of the self-stabilising fragment with **share** is given in Figure 7, defining a class \mathbf{s} of self-stabilising expressions, which may be:

- any expression not containing a **share** or **rep** construct, comprising built-in functions;
- three special forms of **share**-constructs, called *converging*, *acyclic* and *minimising* pattern (respectively), defined by restricting both the syntax and the semantic of relevant functional parameters.

We recall here a brief description of the patterns: for a more detailed presentation, the interested reader may refer to [VAB⁺18]. The semantic restrictions on functions are the following.

Converging (C): A function $\mathbf{f}(\phi, \psi, \bar{\mathbf{v}})$ is said converging iff, for every device δ , its return value is closer to $\psi(\delta)$ than the maximal distance of ϕ to ψ .

Monotonic non-decreasing (M): a stateless⁸ function $\mathbf{f}(\mathbf{x}, \bar{\mathbf{x}})$ with arguments of local type is M iff whenever $\ell_1 \leq \ell_2$, also $\mathbf{f}(\ell_1, \bar{\ell}) \leq \mathbf{f}(\ell_2, \bar{\ell})$.

⁸A function $\mathbf{f}(\bar{\mathbf{x}})$ is *stateless* iff its outputs depend only on its inputs and not on other external factors.

Progressive (P): a stateless function $\mathbf{f}(\mathbf{x}, \bar{\mathbf{x}})$ with arguments of local type is P iff $\mathbf{f}(\ell, \bar{\ell}) > \ell$ or $\mathbf{f}(\ell, \bar{\ell}) = \top$ (where \top denotes the unique maximal element of the relevant type).

Raising (R): a function $\mathbf{f}(\ell_1, \ell_2, \bar{\mathbf{v}})$ is raising with respect to total partial orders $<, \triangleleft$ iff: (i) $\mathbf{f}(\ell, \ell, \bar{\mathbf{v}}) = \ell$; (ii) $\mathbf{f}(\ell_1, \ell_2, \bar{\mathbf{v}}) \geq \min(\ell_1, \ell_2)$; (iii) either $\mathbf{f}(\ell_1, \ell_2, \bar{\mathbf{v}}) \triangleright \ell_2$ or $\mathbf{f}(\ell_1, \ell_2, \bar{\mathbf{v}}) = \ell_1$.

Hence, the three patterns can be described as follows.

Converging: In this pattern, variable \mathbf{x} is repeatedly updated through function $\mathbf{f}^{\mathbf{C}}$ and a self-stabilising value \mathbf{s} . The function $\mathbf{f}^{\mathbf{C}}$ may also have additional (not necessarily self-stabilising) inputs $\bar{\mathbf{e}}$. If the range of the metric granting convergence of $\mathbf{f}^{\mathbf{C}}$ is a well-founded set⁹ of real numbers, the pattern self-stabilises since it gradually approaches the value given by \mathbf{s} .

Acyclic: In this pattern, the neighbourhood's values for \mathbf{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 \mathbf{x}). This is accomplished by the built-in function `nbr1t`, which returns a field of booleans selecting the neighbours with lower argument values, and could be defined as `def nbr1t(x) {nbr{x} < x}`.

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

Minimising: In this pattern, the neighbourhood's values for \mathbf{x} are first increased by a monotonic progressive function $\mathbf{f}^{\mathbf{MP}}$ (possibly depending also on other self-stabilising inputs). As specified above, $\mathbf{f}^{\mathbf{MP}}$ 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 `minHoodLoc(ϕ, ℓ)` (which selects the “minimum” in $\phi[\delta \mapsto \ell]$). Finally, this minimum is fed to the *raising* function $\mathbf{f}^{\mathbf{R}}$ together with the old value for \mathbf{x} (and possibly any other inputs $\bar{\mathbf{e}}$), obtaining a result that is higher than at least one of the two parameters. We assume that the partial orders $<, \triangleleft$ are *noetherian*,¹⁰ 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 $\mathbf{f}^{\mathbf{MP}}$ along every possible path (and the effect of the raising function can be proved to be negligible).

For expressions in the self-stabilising fragment, we can prove that the non-equivalent rewriting preserves the limit behavior, and thus may be safely applied in most cases. Furthermore, the rewriting reduces the number of *full rounds of execution* required for stabilisation.

Definition 4.9 (Full Round of Execution). Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be an augmented event structure and $E_0 \subseteq E$ be a set of events such that whenever $\epsilon' < \epsilon \in E_0$ with $d(\epsilon') = d(\epsilon)$, then $\epsilon' \in E_0$. Define $r : E \rightarrow \mathbb{N}$ as:

$$r(\epsilon) = \begin{cases} 0 & \text{if } \epsilon \in E_0 \\ \min \{r(\epsilon') + 1 \mid \epsilon' \rightsquigarrow \epsilon\} & \text{otherwise} \end{cases}$$

⁹An ordered set is *well-founded* iff it does not contain any infinite descending chain.

¹⁰A partial order is *noetherian* iff it does not contain any infinite ascending chains.

Then, we say that the n -th full round of execution after E_0 comprises all events $\epsilon \in E$ such that $r(\epsilon) = n$. If omitted, we assume E_0 to be the $<$ -closure of the finite set of events ϵ not satisfying the equality in Definition 2.10.

Notice that function r above is weakly increasing on the linear sequence of events on a given device: $r(\epsilon) \leq r(\epsilon') \leq r(\epsilon) + 1$ whenever $\epsilon \rightsquigarrow \epsilon'$ and $d(\epsilon) = d(\epsilon')$.

Theorem 4.10. *Assume that every built-in operator is self-stabilising. Then closed expressions \mathbf{s} as in Figure 7 self-stabilise to the same limit for $\mathbf{rep} + \mathbf{nbr}$ as their rewritings with \mathbf{share} , the latter with a tighter bound on the number of full rounds of execution of a network needed before stabilisation.*

Proof. See Appendix B. □

5. APPLICATION AND EMPIRICAL VALIDATION

Having developed the **share** construct and shown that it should be able to significantly improve the performance of field calculus programs, we have also applied this development by extending the Protelis [PVB15] implementation of field calculus to support **share** (the implementation is a simple addition of another keyword and accompanying implementation code following the semantics expressed above). We have further upgraded every function in the **protelis-lang** library [FPBV17] with an applicable **rep/nbr** combination to use the **share** construct instead, thereby also improving every program that makes use of these libraries of resilient functions. The official Protelis distribution includes these changes to the language and the library into the main distribution, starting with version 11.0.0. To validate the efficacy of both our analysis and its applied implementation, we empirically validate the improvements in performance for a number of these upgraded functions in simulation.

5.1. Evaluation Setup. We experimentally validate the improvements of the **share** construct through two simulation examples. In both, we deploy a number of mobile devices, computing rounds asynchronously at a frequency of 1 ± 0.1 Hz, and communicating within a range of 75 meters. Mobile devices were selected because they pose a further challenge with respect to static ones: in fact, while in a statically deployed system only the transient to stability can be measured, in a dynamic situation the coordination system must cope with continuous, small disruptions by continuously adapting to an evolving situation. All aggregate programs have been written in Protelis [PVB15] and simulations performed in the Alchemist environment [PMV13]. All the results reported in this paper are the average of 200 simulations with different seeds, which lead to different initial device locations, different waypoint generation, and different round frequency. Data generated by the simulator has been processed with Xarray [HH17] and matplotlib [Hun07]. For the sake of brevity, we do not report the actual code in this paper; however, to guarantee complete reproducibility, the execution of the experiments has been entirely automated, and all the resources have been made publicly available along with instructions.¹¹

In the first scenario, we position 2000 mobile devices into a corridor room with sides of, respectively, 200m and 2000m. Two devices are “sources” and are fixed, while the remaining

¹¹ Experiments are separated in two blocks, available on two separate repositories:
<https://bitbucket.org/danyusk/experiment-2019-coordination-aggregate-share/>
<https://github.com/DanySK/Experiment-2019-LMCS-Share>

1998 are free to move within the corridor randomly. We experiment with different locations for the two fixed devices, ranging from the opposite ends of the corridor to a distance of 100m. At every point of time, only one of the two sources is active, switching at 80 seconds and 200 seconds (i.e., the active one gets disabled, the disabled one is re-enabled). Devices are programmed to compute a field yielding everywhere the farthest distance from any device to the current active source. In order to do so, they apply three widely-used general coordination operations [FMSM⁺13, VAB⁺18]: estimation of shortest-path distances, accumulation of values across a region, and broadcast via local spreading. In particular, we use the following specific algorithmic variants:

- (1) devices compute a potential field measuring the distance from the active source through BIS [ADV18] (`bisGradient` routine in `protelis:coord:spreading`);
- (2) devices then accumulate the maximum distance value descending the potential towards the source, through Parametric Weighted Multi-Path C [ABDV19] (an optimized version of `C` in `protelis:coord:accumulation`);
- (3) finally, devices broadcast the accumulated value along the potential, somewhat similar to the chemotaxis coordination pattern [FMSM⁺13], from the source to every other device in the system (an optimized version of the `broadcast` algorithm available in `protelis:coord:spreading`, which tags values from the source with a timestamp and propagates them by selecting more recent values).

The choice of the algorithms to be used in validation is critical. The usage of `share` is able to directly improve the performance of algorithms with solid theoretical guarantees; however, it may also exacerbate errors and instabilities for more ad-hoc algorithms, by allowing them to propagate quicker and more freely, preventing (or slowing down) the stabilization of the algorithm result whenever the network configuration and input is not constant. Of the set of available algorithms for spreading and collecting data, we thus selected variants with smoother recovery from perturbation: optimal single-path distance estimation (BIS gradient [ADV18]), optimal multi-path broadcast [VAB⁺18], and the latest version of data collection (parametric weighted multi-path [ABDV19], fine-tuning the weight function).

We are interested in measuring the error of each step (namely, in distance vs. the true values), together with the lag through which these values were generated (namely, by propagating a time-stamp together with values, and computing the difference with the current time). We call this measurement error *error in distance*, as it indicates how far the distance estimation is from reality. Likewise, we call the measured information lag *error in time*, as it indicates how long it takes for information to flow across the network from the source to other devices. Moreover, we want to inspect how the improvements introduced by `share` accumulate across the composition of algorithms. To do so, we measure the error in two conditions: (i) composite behavior, in which each step is fed the result computed by the previous step, and (ii) individual behavior, in which each step is fed an ideal result for the previous step, as provided by an oracle.

Figure 8 shows the results from this scenario. Observing the behavior of the individual computations, it is immediately clear how the `share`-based version of the algorithm provides faster recovery from network input discontinuities and lower errors at the limit. These effects are exacerbated when multiple algorithms are composed to build aggregate applications. The only counterexample is the limit of distance estimations, for which `rep` is marginally better, with a relative error less than 1% lower than that of `share`.

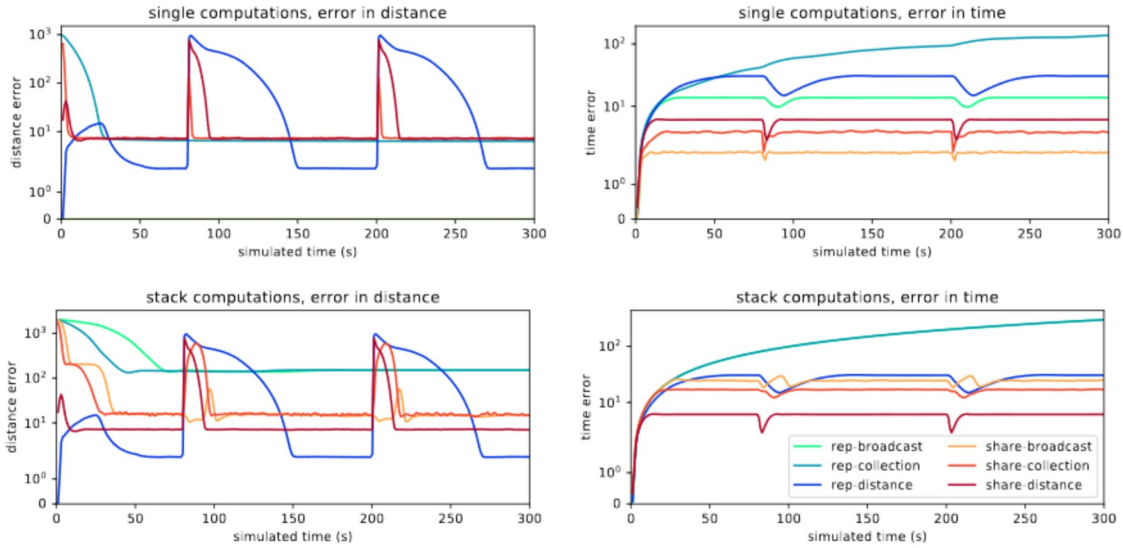


Figure 8: Performance in the corridor scenario, for both individual algorithms (top) and the composite computation (bottom). Vertical axis is linear in $[0, 1]$ and logarithmic above. Charts on the left column show distance error, while the right column shows time error. The versions of the algorithms implemented with `share` (warm colours) produce significantly less error and converge significantly faster in case of large disruptions than with `rep` (cold colours). Peaks at $t=80s$ and $t=200s$ are due to the algorithm re-stabilizing as a consequence of the active source switching between the two opposite nodes.

Moreover, notice that the collection algorithm with `rep` was not able to recover from changes at all, as shown by the linearly increasing delay in time (and the absence of spikes in distance error). The known weakness of multi-path collection strategies, that is, failing to react to changes due to the creation of information loops, proved to be much more relevant and invalidating with `rep` than with `share`.

Further details on the improvements introduced by `share` are depicted in Figure 9, which shows both the lag between two selected devices and how such lag is influenced by the distance between them. Algorithms implemented on `share` provide, as expected, significantly lower network lags, and the effect is more pronounced as the distance between nodes increases: in fact, even though network lags expectedly scale linearly in both cases, `rep`-based versions accumulate lag much more quickly.

In the second example, we deploy 500 devices in a city center, and let them move as though being carried by pedestrians, moving at walking speed ($1.4 \frac{m}{s}$) towards random waypoints along roads open to pedestrian traffic (using map data from OpenStreetMaps [HW08]). In this scenario, devices must self-organize service management regions with a radius of at most 200 meters, creating a Voronoi partition as shown in Figure 10 (functions `S` and `voronoiPartitioningWithMetric` from `protelis:coord:sparsechoice`). We evaluate performance by measuring the number of partitions generated by the algorithm, and the average and maximum node distance error, where the error for a node n measures how far a node is beyond of the maximum boundary for its cluster. This is computed as

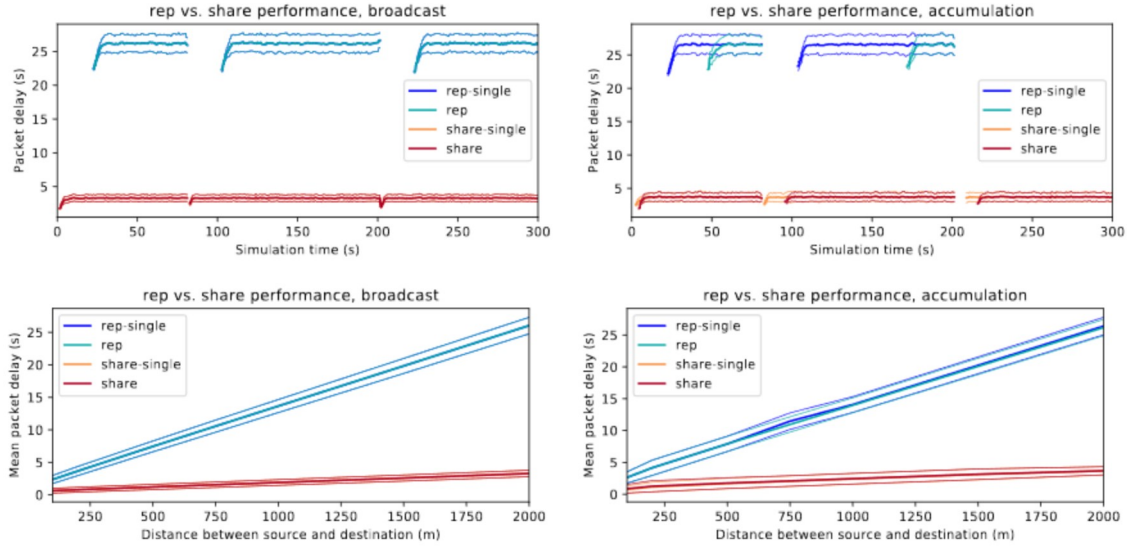


Figure 9: Performance in the corridor scenario, showing on top the packet lag between the two fixed devices for the scenario in which they are at opposite ends of the corridor, and on the bottom how the average packet lag changes with the distance between such devices. Broadcast data is on the left, accumulation on the right. Thinner lines depict mean \pm standard deviation. Darker lines depict “stacked” computations, namely, they use respectively **rep**-based or **share**-based algorithms to compute distances; lighter lines depict “single” computations, where distances are provided by an oracle. The versions of the algorithms implemented with **share** (warm colours) stabilize faster, and once stabilized they provide much lower network lags. The effect stacks when multiple algorithms are used together, as shown by the chart on packet delay in accumulation (top right): the collection algorithm using the distance computed with **rep** requires a longer time for stabilization, after which it provides the same performance (in terms of lag) as the version relying on an oracle. Bottom charts show how both implementations scale linearly with the distance between devices (hence, for a network, linearly in its diameter); however, for **rep**-based algorithms scaling is noticeably worse. Perturbations at $t=80$ s and $t=200$ s are due to the algorithm re-stabilizing as a consequence of the active source switching between the two opposite nodes

$e_n = \max(0, d(n, l_n) - r)$, where d computes the distance between two devices, l_n is the leader for the cluster n belongs to, and r is the maximum allowed radius of the cluster.

Figure 11 shows the results from this scenario, which also confirm the benefits of faster communication with **share**. The algorithm implemented with **share** has much lower error, mainly due to faster convergence of the distance estimates, and consequent higher accuracy in measuring the distance from the partition leader. Simultaneously, it creates a marginally lower number of partitions, by reducing the amount of occasional single-device regions which arise during convergence and re-organization.

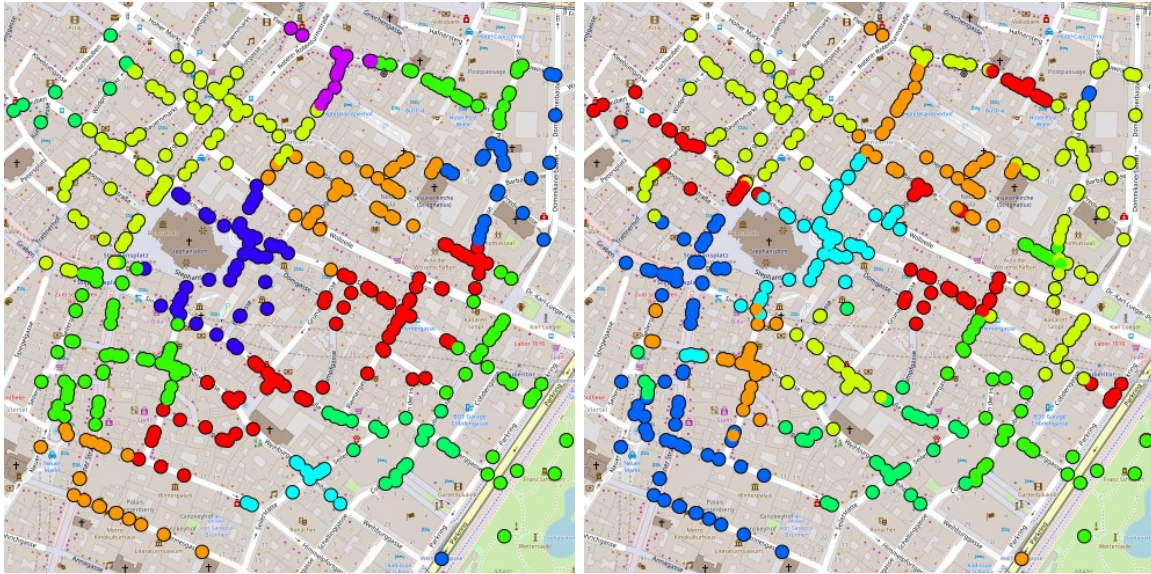


Figure 10: Snapshots of the Voronoi partitioning scenario using **share** (left) or **rep** (right). Colored dots are simulated devices, with each region having a different colour. Faster communication with **share** leads to a higher accuracy in distance estimation, allowing the **share** implementation to perform a better division into regions and preventing regions from expanding beyond their limits: note the mixing of colours on the right.

6. CONCLUSION AND FUTURE WORK

We have introduced a novel primitive for field-based coordination, **share**, allowing declarative expression of unified and coherent operation mechanisms for state-preservation, communication to neighbours, and aggregation of received messages. More specifically, we have shown that this primitive significantly accelerated field calculus programs involving spreading of information, that programs can be automatically rewritten to use **share**, and that transformation to use **share** preserves the key convergence property of self-stabilization. Finally, we have made this construct available for use in applications through an extension of the Protelis field calculus implementation and its accompanying libraries, and have empirically validated the expected improvements in performance through experiments in simulation. Indeed, through this distribution the **share** construct is already being used in industrial applications (e.g., [PDB⁺19, ST20]). In these applications, every use of **rep + nbr** has been replaced by **share**. This replacement has been effected in two ways: first, by use of the new version of the Protelis library and second, by direct conversion of all application code using **rep + nbr** following the speed-improving Rewriting 3 from Section 4.4. Anecdotal reports of system performance from these applications show improvement consistent with the results in this paper. The impact of this work is thus to significantly increase the pragmatic applicability of a wide range of results from aggregate computing.

In future work, we plan to study for which algorithms the usage of **share** may lead to increased instability, thus fine-tuning the choice of **rep** and **nbr** over **share** in the Protelis

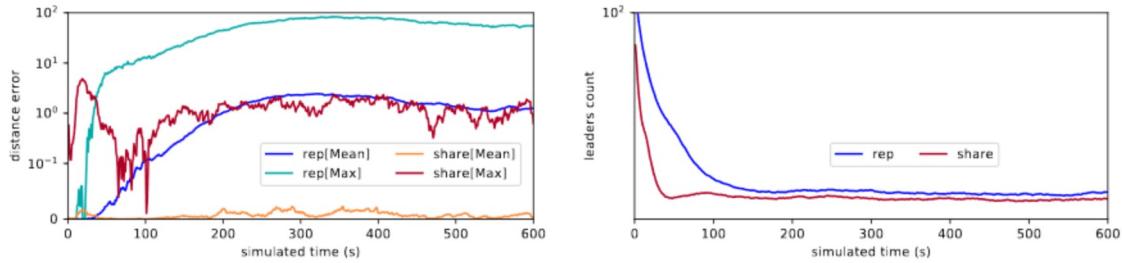


Figure 11: Performance in the Voronoi partition scenario: error in distance on the left, leaders count with time on the right. Vertical axis is linear in $[0, 0.1]$ and logarithmic elsewhere. The version implemented with `share` has much lower error: the mean error is negligible, and the most incorrect value, after an initial convergence phase, is close to two orders of magnitude lower than with `rep`, as faster communication leads to more accurate distance estimates. The leader count shows that the systems create a comparable number of partitions, with the `share`-based featuring faster convergence to a marginally lower number due to increased consistency in partitioning.

library. Furthermore, we intend to fully analyze the consequences of `share` for improvement of space-time universality [ABDV18], self-adaption [BVPD17], real-time properties [ADVB18], and variants of the semantics [ADVC16] of the field calculus. It also appears likely that the field calculus can be simplified by the elimination of both `rep` and `nbr` by finding a mapping by which `share` can also be used to implement any usage of `nbr`. Finally, we believe that the improvements in performance will also have positive consequences for nearly all current and future applications that are making use of the field calculus and its implementations and derivatives. As such, it can also suggest alternative formulations or new operators in other field-based coordination languages, such as [MZ09, WSBC04, VPM⁺15, LLM17, VPB12].

Acknowledgements. We thank the anonymous COORDINATION 2019 referees for their comments and suggestions on improving the presentation.

REFERENCES

- [ABD⁺19] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. The share operator for field-based coordination. In *Coordination Models and Languages*, volume 11533 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2019.
- [ABDV18] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. Space-time universality of field calculus. In *Coordination Models and Languages*, volume 10852 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2018.
- [ABDV19] Giorgio Audrito, Sergio Bergamini, Ferruccio Damiani, and Mirko Viroli. Effective collective summarisation of distributed data in mobile multi-agent systems. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. ACM, 2019.
- [ACDV17] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Compositional blocks for optimal self-healing gradients. In *11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2017)*, pages 91–100. IEEE, 2017.

- [ADV17] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. Optimally-self-healing distributed gradient structures through bounded information speed. In *Coordination Models and Languages*, volume 10319 of *LNCS*, pages 59–77. Springer, 2017.
- [ADV18] Giorgio Audrito, Ferruccio Damiani, and Mirko Viroli. Optimal single-path information propagation in gradient-based algorithms. *Science of Computer Programming*, 166:146–166, 2018.
- [ADVB18] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Enrico Bini. Distributed real-time shortest-paths computations with the field calculus. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 23–34. IEEE Computer Society, 2018.
- [ADVC16] Giorgio Audrito, Ferruccio Damiani, Mirko Viroli, and Roberto Casadei. Run-time management of computation domains in field calculus. In *1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 192–197. IEEE, 2016.
- [ARGL⁺07] 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.
- [AVD⁺19] Giorgio Audrito, Mirko Viroli, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *ACM Transactions on Computational Logic (TOCL)*, 20(1):5:1–5:55, 2019.
- [BB06] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.
- [BDU⁺13] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013.
- [BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 48(9), 2015.
- [But02] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, USA, 2002.
- [BVPD17] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution in the Internet of Things. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(3):12:1–12:29, 2017.
- [CGG⁺05] 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.
- [Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
- [CN03] Lauren Clement and Radhika Nagpal. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, 2003.
- [Coo99] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer*. PhD thesis, MIT, Cambridge, MA, USA, 1999.
- [DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [FMSM⁺13] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluís Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.
- [FPBV17] Matteo Francia, Danilo Pianini, Jacob Beal, and Mirko Viroli. Towards a foundational api for resilient distributed systems design. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 27–32. IEEE, 2017.
- [GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairós. In *Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, 2005.
- [GGMP02] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. Computational models for integrative and developmental biology. Technical Report 72-2002, U. d’Evry, LaMI, 2002.
- [GMCS05] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computations in space and space in computations. In *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 137–152. Springer, Berlin, 2005.

- [HH17] S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017.
- [Hun07] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.
- [HW08] M. Haklay and P. Weber. OpenStreetMap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, oct 2008.
- [IPW01] 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.
- [Kon03] Attila Kondacs. Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 633–638. Morgan Kaufmann Publishers Inc., 2003.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [LLM17] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *Logical Methods in Computer Science*, 13(1), 2017.
- [LMMD88] C. Lasser, J.P. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.
- [MFHH02] 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.*, 36:131–146, 2002.
- [MZ09] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Transactions on Software Engineering Methodologies (TOSEM)*, 18(4):1–56, 2009.
- [Nag01] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Cambridge, MA, USA, 2001.
- [NW04] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks, DMSN '04*, pages 78–87. ACM, 2004.
- [PDB⁺19] Aaron Paulos, Soura Dasgupta, Jacob Beal, Yuanqiu Mo, Khoi Hoang, Lyles J Bryan, Partha Pal, Richard Schantz, Jon Schewe, Ramesh Sitaraman, et al. A framework for self-adaptive dispersal of computing services. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 98–103. IEEE, 2019.
- [PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation*, 7(3):202–215, 2013.
- [PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: Practical aggregate programming. In *ACM Symposium on Applied Computing 2015*, pages 1846–1853, April 2015.
- [ST20] Swarm tactics: A collection of technologies for developing, simulating and executing swarm tactics at scale. <http://www.swarmtactics.com/>, 2020. Accessed: April 3, 2020.
- [VAB⁺18] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Transactions on Modelling and Computer Simulation (TOMACS)*, 28(2):16:1–16:28, 2018.
- [Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
- [VBD⁺19] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *Journal of Logical and Algebraic Methods in Programming*, 109, 2019. Article 100486.
- [VPB12] Mirko Viroli, Danilo Pianini, and Jacob Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In Marjan Sirjani, editor, *Coordination Languages and Models*, volume 7274 of *LNCS*, pages 212–229. Springer-Verlag, June 2012. Proceedings of the 14th Conference of Coordination Models and Languages (Coordination 2012), Stockholm (Sweden), 14-15 June.
- [VPM⁺15] 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.

- [WSBC04] 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.
- [Yam07] Daniel Yamins. *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*. PhD thesis, Harvard, Cambridge, MA, USA, 2007.
- [YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31:9–18, 2002.

APPENDIX A. PROOF OF TCNS COMPLETENESS

In this section, we prove that the TCNS is able to capture the message passing details of any augmented event structure.

Restatement of Theorem 3.5 (TCNS Completeness). *Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be an augmented event structure. Then there exist (infinitely many) system evolutions following \mathbf{E} .*

Proof. Define a set $T = \{\epsilon^c \mid \epsilon \in E\} \cup \{\epsilon^s \mid \epsilon \in E\}$, including two elements ϵ^c, ϵ^s for every event ϵ (representing the *computation* and *send* phase of the event). Define \rightsquigarrow on T as:

- (1) $\epsilon_1^s \rightsquigarrow \epsilon_2^c$ for each pair of neighbour events $\epsilon_1 \rightsquigarrow \epsilon_2$;
- (2) $\epsilon_1^c \rightsquigarrow \epsilon_2^s$ for each pair of time-dependent events $\epsilon_1 \dashrightarrow \epsilon_2$;¹²
- (3) $\epsilon^c \rightsquigarrow \epsilon^s$ for each event $\epsilon \in E$.

First, we prove that the \rightsquigarrow relation on T is acyclic due to the *immediacy* property. Notice that \rightsquigarrow always alternates between *computation* and *send* elements of T , and in a chain of \rightsquigarrow every other transition must be of type (1). Suppose then by contradiction that $\epsilon_1^s \rightsquigarrow \epsilon_2^c \rightsquigarrow \dots \rightsquigarrow \epsilon_{2n}^c \rightsquigarrow \epsilon_1^s$ is a cycle in T . If no transition of type (2) is present, the cycle in T corresponds to a cycle $\epsilon_2 \rightsquigarrow \epsilon_4 \rightsquigarrow \dots \rightsquigarrow \epsilon_{2n} \rightsquigarrow \epsilon_2$ in E which is a contradiction. Then some transitions of type (2) must be present: assume they are $\epsilon_{2k_i}^c \rightsquigarrow \epsilon_{2k_i+1}^s$ corresponding to $\epsilon_{2k_i} \dashrightarrow \epsilon_{2k_i+1}$ for $i \leq m$ and $m \leq n$, $k_i \leq n$ increasing. Then $\epsilon_{2k_i+1}^s \rightsquigarrow \dots \rightsquigarrow \epsilon_{2k_i+1}^c$ corresponds to a chain $\epsilon_{2k_i+1} \rightsquigarrow \dots \rightsquigarrow \epsilon_{2k_i+1}$ in E , hence in particular $\epsilon_{2k_i+1}^s < \epsilon_{2k_i+1}^c$. Thus $\epsilon_{2k_1} \dashrightarrow \epsilon_{2k_1+1} < \epsilon_{2k_2} \dashrightarrow \dots < \epsilon_{2k_1}$ is a cyclic sequence contradicting *immediacy*, concluding the proof of the claim that \rightsquigarrow is acyclic on T .

Since \rightsquigarrow is acyclic on T , there exists at least one ordering of $T = \langle \epsilon_1^{x_1}, \dots, \epsilon_\ell^{x_\ell} \rangle$ compatible with \rightsquigarrow , i.e. such that $\epsilon_i^{x_i} \rightsquigarrow \epsilon_j^{x_j} \Rightarrow i < j$. Define by induction a system evolution \mathcal{S}_i for $i \leq \ell$ translating the elements of T (in order), starting from the empty system evolution without transitions $\mathcal{S}_0 = \langle \emptyset, \emptyset; \emptyset, \emptyset \rangle$.

Consider a step $i \leq \ell$ and let $\delta_i = d(\epsilon_i)$. If $x_i = c$ (we are at a *computation* element of T), add the following two transitions $\mathcal{S}_i = \mathcal{S}_{i-1} \xrightarrow{env} N' \xrightarrow{\delta_i+} N''$:

- first, an *env* transition inserting δ_i into the domain of the final system configuration in \mathcal{S}_{i-1} (if not already present);
- then, a δ_i+ transition representing the computation, where the filter F clears out from the value-tree environment $\Psi(\delta_i)$ the value trees corresponding to devices not in $X = \{d(\epsilon') \mid \epsilon' \rightsquigarrow \epsilon_i\}$.

If $x_i = s$ (we are at a *send* element of T), add the following three transitions to the system $\mathcal{S}_i = \mathcal{S}_{i-1} \xrightarrow{env} N' \xrightarrow{\delta_i-} N'' \xrightarrow{env} N'''$:

- first, an *env* transition setting $\tau(\delta_i)$ to $Y = \{d(\epsilon') \mid \epsilon_i \rightsquigarrow \epsilon'\}$, possibly adding devices in Y to the domain of the system configuration if not already present;
- secondly, a δ_i- transition;
- finally, another *env* transition, which removes δ_i from the domain of the system configuration if $\text{next}(\epsilon_i)$ does not exist, or it does nothing if $\text{next}(\epsilon_i)$ exists.

Then, the system evolution \mathcal{S}_ℓ follows \mathbf{E} (c.f. Definition 3.4). Notice that many system evolutions may follow \mathbf{E} : besides the existence of many different linearisations of T according to \rightsquigarrow , *env* transitions can be added in an unbounded number of ways. \square

¹²We recall that $\epsilon_1 \dashrightarrow \epsilon_2$ iff $\epsilon_2 \rightsquigarrow \text{next}(\epsilon_1)$ and $\epsilon_2 \not\rightsquigarrow \epsilon_1$ (c.f. Definition 2.3).

APPENDIX B. PROOF OF SELF-STABILISATION

In this section, we prove Theorem 4.10. First, we prove the result for the minimising pattern (Lemma B.1), since it is technically more involved than the proof for the remainder of the fragment. We then prove a stronger form of the desired result (Lemma B.2) more suited for inductive reasoning, which in turn implies Theorem 4.10.

Given a closed self-stabilising expression \mathbf{s} , we denote with $\llbracket \mathbf{s} \rrbracket = \Psi = \bar{\delta} \mapsto \bar{\mathbf{v}}$ the self-stabilising limit value of this expression in a given network graph \mathbf{G} (c.f. Definition 4.8), attained for every system evolution \mathcal{S} of a network following an \mathbf{E} with limit \mathbf{G} . Let:

$$\begin{aligned} \mathbf{s}_{\min}^r &= \text{rep}(\mathbf{e})\{(x) \Rightarrow \mathbf{f}^R(\text{minHoodLoc}(\mathbf{f}^{\text{MP}}(\text{nbr}\{x\}, \bar{\mathbf{s}}^r), \mathbf{s}^r), x, \bar{\mathbf{e}})\} \\ \mathbf{s}_{\min}^s &= \text{share}(\mathbf{e})\{(x) \Rightarrow \mathbf{f}^R(\text{minHoodLoc}(\mathbf{f}^{\text{MP}}(x, \bar{\mathbf{s}}^s), \mathbf{s}^s), \text{localHood}(x), \bar{\mathbf{e}})\} \end{aligned}$$

be corresponding minimising patterns such that $\llbracket \bar{\mathbf{s}}^r \rrbracket = \llbracket \bar{\mathbf{s}}^s \rrbracket = \bar{\Psi}$, $\llbracket \mathbf{s}^r \rrbracket = \llbracket \mathbf{s}^s \rrbracket = \Psi$. Let $P = \bar{\delta}$ be a path in the network (a sequence of pairwise connected devices), and define its *weight* as the result of picking the eventual value $\ell_1 = \Psi(\delta_1)$ of \mathbf{s}^r in the first device δ_1 , and repeatedly passing it to subsequent devices through the monotonic progressive function, so that $\ell_{i+1} = \mathbf{f}^{\text{MP}}(\ell_i, \bar{\mathbf{v}})$ where $\bar{\mathbf{v}}$ is the result of projecting fields in $\bar{\Psi}(\delta_{i+1})$ to their δ_i component (leaving local values untouched). Notice that the weight is well-defined since function \mathbf{f}^{MP} is required to be stateless. Finally, let Ψ_{out} be such that $\Psi_{\text{out}}(\delta) = \ell_\delta$ is the minimum weight for a path P ending in δ .

Lemma B.1. *Let $\mathbf{s}_{\min}^r, \mathbf{s}_{\min}^s$ be corresponding minimising patterns, whose sub-expressions stabilise within n^r, n^s full rounds of execution (respectively) with $n^r \geq n^s$. Then they both stabilise to Ψ_{out} , with a bound on the number of full rounds of execution which is greater for \mathbf{s}_{\min}^r than for \mathbf{s}_{\min}^s .*

Proof. Let ℓ_δ be the minimal weight for a path P ending in δ , and let $\delta^0, \delta^1, \dots$ 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 \mathbf{f}^{MP} is progressive.

Let \mathcal{S} be a system evolution following $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ with limit \mathbf{G} . We now prove by complete induction on i that after a certain number of full rounds of execution n_i^r, n_i^s expressions $\mathbf{s}_{\min}^r, \mathbf{s}_{\min}^s$ stabilise to ℓ_{δ^i} in device δ^i and assume values $\geq \ell_{\delta^i}$ in devices δ^j with $j \geq i$.

By inductive hypothesis, assume that devices δ^j with $j < i$ are all self-stabilised from a certain number of full rounds of execution n_{i-1}^r, n_{i-1}^s . Thus, their limit values are available to neighbours after $n_{i-1}^r + 2, n_{i-1}^s + 1$ full rounds of execution respectively. Consider the evaluation of the expressions $\mathbf{s}_{\min}^r, \mathbf{s}_{\min}^s$ in a device δ^k with $k \geq i$. Since the local argument ℓ of 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 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 whole $\mathbf{s}_{\min}^r, \mathbf{s}_{\min}^s$ expression among all devices δ^j with $j \geq i$ (delayed by one round for $\text{rep} + \text{share}$), since \mathbf{f}^{MP} is progressive.

It follows that as long as the minimum value for the whole expressions among non-stable devices is lower than ℓ_{δ^i} , the result of the `minHoodLoc` subexpression is *strictly greater* than this minimum value. The same holds for the overall value, since it is obtained by combining the output of `minHoodLoc` with the previous value for \mathbf{x} through the rising function \mathbf{f}^R , and a rising function has to be equal to the first argument (the `minHoodLoc` result strictly greater than the minimum), or \triangleright than the second. In the latter case, it also needs to be greater or equal to the first argument (again, strictly greater than the minimum) or strictly greater than the second argument¹³ (not below the minimum value).

Thus, every full round of execution (two full rounds for `rep + nbr`, in order to allow value changes to be received) the minimum value among non-stable devices has to increase, until it eventually surpasses ℓ_{δ^i} since $<$ is noetherian. This happens within at most $n_{i-1}^r + 2x$, $n_{i-1}^s + x$ full rounds of execution respectively, where x is the length of the longest increasing sequence between $\ell_{\delta^{i-1}}$ and ℓ_{δ^i} (longest sequence up to ℓ_{δ^i} if $i = 0$). From that point on, that minimum cannot drop below ℓ_{δ^i} , and the output of `minHoodLoc` in δ^i stabilises to ℓ_{δ^i} . In fact, if P is 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 `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 \mathbf{f}^{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 `minHoodLoc` operator) is exactly ℓ_{δ^i} .

Since the order \triangleleft is noetherian, the rising function on δ^i has to select its first argument in a number of rounds y at most equal to the longest increasing sequence from ℓ_{δ^i} . Thus, it will select the output of the `minHoodLoc` subexpression, which is ℓ_{δ^i} , after $n_{i-1}^r + 2x + y$, $n_{i-1}^s + x + y$ full rounds of execution. From that point on, the minimising expression will have self-stabilised on device δ^i to ℓ_{δ^i} , and every device δ^j with $j \geq i$ will attain values $\geq \ell_{\delta^i}$, concluding the inductive step and the proof. \square

Let Ψ be a computational field. We write $\mathbf{s}[\mathbf{x} := \Psi]$ to indicate an aggregate process in which each device is computing a possibly different substitution $\mathbf{s}[\mathbf{x} := \Psi(\delta)]$ of the same expression.

Lemma B.2. *Assume that every built-in operator is self-stabilising. Let \mathbf{s}^r be an expression in the self-stabilising fragment of [VAB⁺18], \mathbf{s}^s its non-equivalent translation with `share`, and $\bar{\Psi}$ be a sequence of computational fields on \mathbf{G} of the same length as the free variables $\bar{\mathbf{x}}$ occurring in $\mathbf{s}^r, \mathbf{s}^s$. Then $\mathbf{s}^r[\bar{\mathbf{x}} := \bar{\Psi}]$, $\mathbf{s}^s[\bar{\mathbf{x}} := \bar{\Psi}]$ self-stabilise to the same limit, and the second does so with a smaller bound on the number of full rounds of execution.*

Proof. Let \mathcal{S} be a system evolution following $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ with limit \mathbf{G} . The proof proceeds by induction on the syntax of expressions and programs. The given expressions $\mathbf{s}^r, \mathbf{s}^s$ could be:

- A variable \mathbf{x}_i , so that $\mathbf{s}^r[\bar{\mathbf{x}} := \bar{\Psi}] = \mathbf{s}^s[\bar{\mathbf{x}} := \bar{\Psi}] = \Psi_i$ are already self-stabilised and identical.
- A value \mathbf{v} , so that $\mathbf{s}^r[\bar{\mathbf{x}} := \bar{\Psi}] = \mathbf{s}^s[\bar{\mathbf{x}} := \bar{\Psi}] = \mathbf{v}$ are already self-stabilised and identical.
- A `let`-expression `let $\mathbf{x} = \mathbf{s}_1^r$ in \mathbf{s}_2^r , let $\mathbf{x} = \mathbf{s}_1^s$ in \mathbf{s}_2^s` . By inductive hypothesis, the sub-expressions $\mathbf{s}_1^r, \mathbf{s}_1^s$ stabilise to Ψ within $n_1^r \geq n_1^s$ full rounds of execution. After that, `let $\mathbf{x} = \mathbf{s}_1$ in \mathbf{s}_2` evaluates to the same value as the expression $\mathbf{s}_2[\mathbf{x} := \Psi]$ which is

¹³It cannot be equal to the second argument, as it is \triangleright -greater than it.

self-stabilising by inductive hypothesis in a number of full rounds of execution $n_2^r \geq n_2^s$. Thus, the whole `let`-expression stabilises within $n_1^r + n_2^r \geq n_1^s + n_2^s$ full rounds of execution.

- A functional application $\mathbf{f}^r(\bar{\mathbf{s}}^r)$, $\mathbf{f}^s(\bar{\mathbf{s}}^s)$. By inductive hypothesis, all expressions $\bar{\mathbf{s}}^r$, $\bar{\mathbf{s}}^s$ self-stabilise to $\bar{\Psi}$ after a certain amount of full rounds of execution (lower for $\bar{\mathbf{s}}^s$). After stabilisation of the arguments, if $\mathbf{f}^r = \mathbf{f}^s = \mathbf{f}$ is a built-in function then $\mathbf{f}(\bar{\mathbf{s}}^r)$, $\mathbf{f}(\bar{\mathbf{s}}^s)$ stabilises by the assumption on built-ins with the same number of additional full rounds of execution. Otherwise, $\mathbf{f}^r(\bar{\mathbf{s}}^r)$, $\mathbf{f}^s(\bar{\mathbf{s}}^r)$ evaluate to the same value of the expression $\mathit{body}(\mathbf{f}^r)[\mathit{args}(\mathbf{f}^r) := \bar{\Psi}]$ (resp. with \mathbf{f}^s) which are self-stabilising in a number of full rounds of executions lower for \mathbf{f}^s by inductive hypothesis.
- A conditional $\mathbf{s}^r = \mathbf{if}(\mathbf{s}_1^r)\{\mathbf{s}_2^r\}\{\mathbf{s}_3^r\}$, $\mathbf{s}^s = \mathbf{if}(\mathbf{s}_1^s)\{\mathbf{s}_2^s\}\{\mathbf{s}_3^s\}$. By inductive hypothesis, expressions \mathbf{s}_1^r , \mathbf{s}_1^s self-stabilise to Ψ_{guard} (with fewer rounds for share). Let $\mathbf{G}_{\mathit{true}}$ be the sub-graph consisting of devices δ such that $\Psi_{\mathit{guard}}(\delta) = \mathit{true}$, and analogously $\mathbf{G}_{\mathit{false}}$. Assume that \mathbf{s}_2^r , \mathbf{s}_2^s self-stabilise to Ψ_{true} in $\mathbf{G}_{\mathit{true}}$ and \mathbf{s}_3^r , \mathbf{s}_3^s to Ψ_{false} in $\mathbf{G}_{\mathit{false}}$ (with fewer rounds for share). Since a conditional is computed in isolation in the above defined sub-environments, \mathbf{s}^r , \mathbf{s}^s self-stabilise to $\Psi = \Psi_{\mathit{true}} \cup \Psi_{\mathit{false}}$ (with fewer rounds for share).
- A neighbourhood field construction $\mathbf{nbr}\{\mathbf{s}^r\}$, $\mathbf{nbr}\{\mathbf{s}^s\}$. By inductive hypothesis, expressions \mathbf{s}^r , \mathbf{s}^s self-stabilise to Ψ after some rounds of computation (fewer for share). Then $\mathbf{nbr}\{\mathbf{s}^r\}$, $\mathbf{nbr}\{\mathbf{s}^s\}$ self-stabilise to the corresponding Ψ' after one additional full round of execution, where $\Psi'(\delta)$ is Ψ restricted to $\mathcal{N}(\delta)$.
- A converging pattern \mathbf{s}_c^r , \mathbf{s}_c^s :

$$\begin{aligned} \mathbf{s}_c^r &= \mathbf{rep}(\mathbf{e})\{(x) \Rightarrow \mathbf{f}^C(\mathbf{nbr}\{x\}, \mathbf{nbr}\{\mathbf{s}^r\}, \bar{\mathbf{e}})\} \\ \mathbf{s}_c^s &= \mathbf{share}(\mathbf{e})\{(x) \Rightarrow \mathbf{f}^C(x, \mathbf{nbr}\{\mathbf{s}^s\}, \bar{\mathbf{e}})\} \end{aligned}$$

By inductive hypothesis, \mathbf{s}^r , \mathbf{s}^s self-stabilise (the latter with fewer rounds) to a same Ψ . Given any index n , let d_n^r , d_n^s be the maximum distances $\mathbf{s}_c^r - \Psi(d(\epsilon))$, $\mathbf{s}_c^s - \Psi(d(\epsilon))$ realised during events ϵ of the n -th full round of execution.

We prove that d_n^s is strictly decreasing with n , while $d_n^r \geq d_{n-1}^r$, $d_n^r > d_{n+2}^r$ strictly decreases every two rounds. Since distances are computed on a well-founded set, it will follow that they will become zero for a sufficiently large n (smaller for share), thus \mathbf{s}_c^r , \mathbf{s}_c^s stabilise as well to the same Ψ (with fewer rounds for share).

Consider an event on the n -th full round of execution. Thus, neighbours events belong to rounds of execution $\geq n-1$, hence their distance with Ψ is at most d_{n-2}^r , d_{n-1}^s respectively. It follows that the output of the converging function \mathbf{f}^C must be strictly closer to Ψ than d_{n-2}^r , d_{n-1}^s respectively, concluding the proof.

- An acyclic pattern \mathbf{s}_a^r , \mathbf{s}_a^s :

$$\begin{aligned} \mathbf{s}_a^r &= \mathbf{rep}(\mathbf{e})\{(x) \Rightarrow \mathbf{f}^r(\mathbf{mux}(\mathbf{nbr}\mathit{lt}(\mathbf{s}_p^r), \mathbf{nbr}\{x\}, \mathbf{s}^r), \bar{\mathbf{s}}^r)\} \\ \mathbf{s}_a^s &= \mathbf{share}(\mathbf{e})\{(x) \Rightarrow \mathbf{f}^s(\mathbf{mux}(\mathbf{nbr}\mathit{lt}(\mathbf{s}_p^s), x, \mathbf{s}^s), \bar{\mathbf{s}}^s)\} \end{aligned}$$

By inductive hypothesis, \mathbf{s}^r , \mathbf{s}^s self-stabilise (the latter with fewer rounds) to a same Ψ , and similarly for \mathbf{s}_p^r , \mathbf{s}_p^s with Ψ_p and $\bar{\mathbf{s}}^r$, $\bar{\mathbf{s}}^s$ with $\bar{\Psi}$.

Let ϵ be any firing in the first full round of execution (after stabilisation of sub-expressions) of the device δ_0 of minimal potential $\Psi_p(\delta_0)$ in the network. Since $\Psi_p(\delta_0)$ is minimal, $\mathbf{nbr}\mathit{lt}(\mathbf{s}_p^r)$, $\mathbf{nbr}\mathit{lt}(\mathbf{s}_p^s)$ are false and the `mux`-expression reduces to \mathbf{s}^r , \mathbf{s}^s and the whole \mathbf{s}_a^r , \mathbf{s}_a^s to $\mathbf{f}^r(\mathbf{s}^r, \bar{\mathbf{s}}^r)$, $\mathbf{f}^s(\mathbf{s}^s, \bar{\mathbf{s}}^s)$, which self-stabilises by inductive hypothesis (with fewer rounds for share).

Let now ϵ be any firing in the first (second for **rep**) full round of execution after stabilisation of δ_0 of the device δ_1 of second minimal potential $\Psi_p(\delta_1)$. Then the mux-expression in δ_1 only (possibly) depends on the value of the device of minimal potential, which is already self-stabilised and available to neighbours. Thus by inductive hypothesis $\mathbf{s}_a^r, \mathbf{s}_a^s$ self-stabilises also in δ_1 (with fewer rounds for share). By repeating the same reasoning on all devices in order of increasing potential, we obtain a final number of rounds (smaller for share) after which all devices have self-stabilised.

- A minimising **rep**: this case is proved for closed expressions in Lemma B.1, and its generalisation to open expressions is straightforward. \square