The `share` operator for field-based coordination*

Giorgio Audrito[1,2], Jacob Beal[3], Ferruccio Damiani[1,2], Danilo Pianini[4], and Mirko Viroli[4]

[1] Dipartimento di Informatica, University of Torino, Torino, Italy
[2] Centro di Competenza per il Calcolo Scientifico, University of Torino, Torino, Italy
{giorgio.audrito,ferruccio.damiani}@unito.it
[3] Raytheon BBN Technologies, Cambridge (MA), USA
jakebeal@ieee.org
[4] DISI, University of Bologna, Cesena, Italy
{danilo.pianini,mirko.viroli}@unibo.it

Abstract. Recent work in the area of coordination models and collective adaptive systems promotes a view of distributed computations as functions manipulating computational fields (data structures spread over space and evolving over time), and introduces the field calculus as a formal foundation for field computations. In field calculus, evolution (time) and neighbor interaction (space) are handled by separate functional operators: however, this 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 neighbors' values; *(ii)* reduction to a single local value; and *(iii)* update and converse sharing to neighbors of a local variable. 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 a number of frequently used network propagation and collection algorithms.

Keywords: Aggregate Programming · Computational Field · Information Propagation Speed · Spatial Computing

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 [8] is one promising such approach, providing a layered architecture in which programmers can describe computations in terms

* 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.

resilient operations on "aggregate" data structures with values spread over space and evolving in time.

The foundation of this approach is field computation, formalised by the field calculus [32], 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 [5]. Traditionally, in this approach each construct and reusable component is a pure function from fields to fields—a fields is a map from a set of space-time computational events to a set of values—and each primitive construct handles just one key aspect of computation: hence, one construct deals with time (i.e, `rep`, providing field evolution) and one with space (i.e., `nbr`, handling neighbor interaction). However, in recent work on the universality of field calculus, we have identified that the combination of time evolution and neighbor interaction operators in the original field calculus induces a delay, limiting the speed of information propagation that can be achieved efficiently [2].

In this paper, we address this limitation by extending field calculus with the `share` construct, combining time evolution and neighbor interaction into a single new coordination operator that simultaneously implements: *(i)* observation of neighbors' values; *(ii)* reduction to a single local value; and *(iii)* update and converse sharing to neighbors of a local variable.

Following a review of field calculus and its motivating context in Section 2, we introduce the `share` construct in Section 3, empirically validate the predicted acceleration of speed in frequently used network propagation and collection algorithms in Section 4, and conclude with a summary and discussion of future work in Section 5.

2 Background and Motivation

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 [7]: 1) "device-abstraction" methods that abstract and simplify the programming of individual devices and interactions (e.g., TOTA [25], Hood [34], chemical models [33], "paintable computing" [9], Meld [1]) or entirely abstract away the network (e.g., BSP [31], MapReduce [14], Kairos [18]). 2) spatial patterning languages that focus on geometric or topological constructs (e.g., Growing Point Language [12], Origami Shape Language [27], self-healing geometries [11,22], cellular automata patterning [35]), 3) information summarization languages that focus on collection and routing of information (e.g., TinyDB [24], Cougar [36], TinyLime [13], and Regiment [28]), and 4) general purpose space-time computing models (e.g., StarLisp [23], MGS [16,17], Proto [6], aggregate programming [8]).

The field calculus [32,5] belongs to the last of these classes, the general purpose models. Like other core calculi, such as λ-calculus [10] or π-calculus [26],

the field calculus was designed to provide a minimal, mathematically tractable model of computation—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 [2] 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).

That work also, however, identified a key limitation of the current formulation of field calculus: the operators for time evolution and neighbor interaction in field calculus interact such that for most programs either 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 from [2]: Section 2.1 introduces the underlying space-time computational model used by field calculus, Section 2.2 provides a review of field calculus itself, and Section 2.3 explains and illustrates the problematic interaction between time evolution and neighbor interaction operators that will be addressed by the `share` operator in the next section.

2.1 Space-Time Computation

Field calculus considers a computational model in which a program `P` is periodically and asynchronously executed on every device δ. In every round of execution, each device:

1. collects information from sensors, local memory, and the most recent messages from neighbors,[5] the latter in the form of a *neighboring value* map $\phi : \delta \to \mathtt{v}$ from neighbors to values,
2. evaluates program `P` with the information collected as its input,
3. stores the results of the computation locally, as well as broadcasting it to neighbors and possibly feeding it to actuators, and
4. sleeps until it is awaken at the next activation.

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 [2]:

Definition 1 (Event Structure). *An* event structure $\mathbf{E} = \langle E, \rightsquigarrow, < \rangle$ *is a countable set of events* E *together with a neighboring 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* $\{\epsilon' \in E | \epsilon' < \epsilon\}$ *is finite for all* ϵ *(i.e.,* $<$ *is locally finite).*

[5] Stale messages may expire after some timeout.

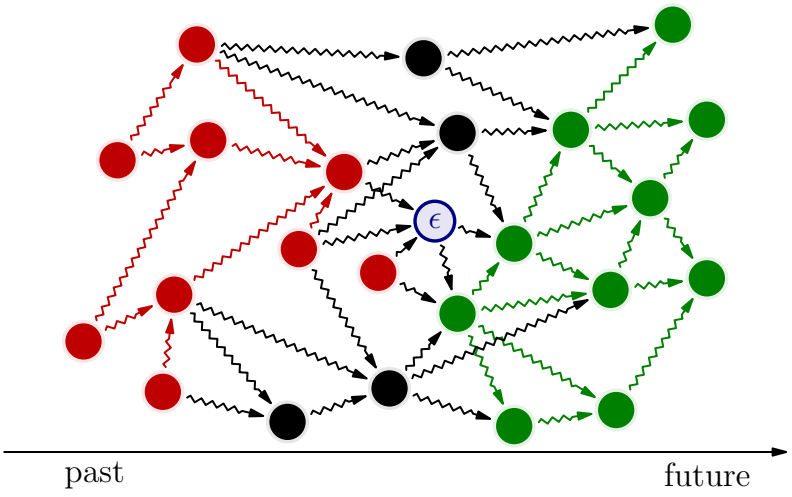

Fig. 1. Example of a space-time event structure, comprising events (circles) and neighbor relations (arrows). Colors indicate causal structure with respect to event ϵ, splitting events into causal past (red), causal future (green) and concurrent (non-ordered, in black). Figure adapted from [2].

Figure 1 shows an example of such an event structure, showing how these relations partition events into "causal past", "causal future", and non-ordered "concurrent" subspaces with respect to any given event ϵ. 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 neighbor to the head neighbor. 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 2.3, however, this is problematic in field calculus as it has been previously defined.

Our aggregate constructs are then space-time data values that map an event structure to a value for each event:

Definition 2 (Space-Time Value). *Let V be any domain of computational values and $\mathbf{E}$ be a given event structure. A space-time value $\Phi = \langle \mathbf{E}, f \rangle$ is a pair comprising the space and a function $f : E \to V$ that maps the events E of $\mathbf{E}$ to values.*

$\texttt{P} ::= \overline{\texttt{F}}\ \texttt{e}$	program
$\texttt{F} ::= \texttt{def d}(\overline{\texttt{x}})\ \{\texttt{e}\}$	function declaration
$\texttt{e} ::= \texttt{x} \mid \texttt{v} \mid \texttt{f}(\overline{\texttt{e}}) \mid \texttt{if}(\texttt{e})\{\texttt{e}\}\{\texttt{e}\} \mid \texttt{nbr}\{\texttt{e}\} \mid \texttt{rep}(\texttt{e})\{(\texttt{x})\ \texttt{=>}\ \texttt{e}\}$	expression
$\texttt{f} ::= \texttt{d} \mid \texttt{b}$	function name
$\texttt{v} ::= \ell \mid \phi$	value
$\ell ::= \texttt{c}(\overline{\ell})$	local value
$\phi ::= \overline{\delta} \mapsto \overline{\ell}$	neighboring field value

Fig. 2. Abstract syntax of the field calculus, adapted from [32]

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

2.2 Field Calculus

Field calculus is a tiny universal language for computation of space-time values. Figure 2 gives an abstract syntax for field calculus based on the presentation in [32] (covering a subset of the higher-order field calculus in [5], but including all of the issues addressed by the `share` construct). In this syntax, the overbar notation $\overline{\texttt{e}}$ indicates a sequences of elements (e.g., $\overline{\texttt{e}}$ stands for $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n$), and multiple overbars are expanded together (e.g., $\overline{\delta} \mapsto \overline{\ell}$ stands for $\delta_1 \mapsto \ell_1, \delta_2 \mapsto \ell_2, \ldots, \delta_n \mapsto \ell_n$). 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 constructs of field calculus that are the focus of this paper, respectively responsible for evolution of state over time and for sharing information between neighbors.

A field calculus program `P` is a set of function declarations $\overline{\texttt{F}}$ 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 $\overline{\ell}$), such as a Boolean, number, string, pair, tuple, etc;
  - A *neighboring (field) value* ϕ that associates neighbor devices δ to local values ℓ, e.g., a map of neighbors to the distances to those neighbors.
- A function call $\texttt{f}(\overline{\texttt{e}})$ 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 computation of e_2 in the former area, and e_3 in the latter.
– The `nbr`\{`e`\} construct, which creates a neighboring field value mapping neighbors to their latest available result of evaluating `e`. In particular, each device δ:
  1. shares its value of `e` with its neighbors, and
  2. evaluates the expression into a neighboring field value ϕ associating to each neighbor δ' of δ the latest evaluation of `e` shared from δ'.

  Note that within an `if` branch, sharing is restricted to being between device events within the subspace of the branch.
– The `rep`(e_1)\{(`x`) `=>` e_2\} construct, which models state evolution over time, initializing the value of `x` to e_1 and evolving that value in every execution through evaluation of expression e_2.

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

```
def distanceTo(source) {
  rep (infinity) { (d) =>
    if(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 neighbor field value (built-in function `minHood`) of the distance to each neighbor (built-in function `nbrRange`) plus that neighbor's distance estimate `nbr{d}`

Additional illustrative examples and full mathematical details of these constructs and the formal semantics of their evaluation can be found in [32].

2.3 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 [2]. 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 neighboring field 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 neighbors. Once old becomes true, it stays true for the rest of the computation. Notice, however, that a neighboring 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 neighbors 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 neighbors, but does not solve the problem for neighbors of neighbors, which still have to wait an additional round before old is updated.

In fact, it appears that the only way to avoid delays at some depth of neighbor relations is by using unbounded recursion, as previously outlined in [2]:

```
def ever3(condition) {
  rep (false) { (old) =>
    if (countHood() == 0) { old || condition } {
      ever3(anyHoodPlusSelf(nbr{old || condition}))
} } }
```

where countHood counts the number of neighbors, i.e., determining whether any neighbor has reached the same depth of recursion in the branch. Thus, in ever3, neighbors' values of cond 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 $\epsilon_0 \rightsquigarrow \ldots \rightsquigarrow \epsilon$ 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 [2]. 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.

3 The Share Construct

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

$$\texttt{share}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \texttt{e}_2\}$$

While the syntax of this new `share` construct is identical to that of `rep`, the two constructs differ in the way the construct variable `x` is interpreted each round:

- in `rep`, the value of `x` is the value produced by evaluating the construct in the previous round, or the result of evaluating $\texttt{e}_1$ if there is no prior-round value;
- in `share`, on the other hand, `x` is a *neighboring field* comprising that same value for the current device plus any values of the construct produced by neighbors in their most recent evaluation.

Notice that since `x` is a neighboring field rather than a local value, $\texttt{e}_2$ is responsible for processing it into a local value that can be shared with neighbors at the end of the evaluation. Furthermore, notice that the value for δ in the field `x` corresponds exactly to the value that would be substituted in `x` for a corresponding `rep` construct. Thus, a `rep` construct may as well be equivalently rewritten as a `share` construct as follows:

$$\texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \texttt{e}_2\} \longrightarrow \texttt{share}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \texttt{e}_2[\texttt{x} := \texttt{localHood}(\texttt{x})]\}$$

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

Whenever a field calculus program used `x` only as $\texttt{nbr}\{\texttt{x}\}$ inside the $\texttt{e}_2$ expression of a `rep`, however, the `share` construct can improve over `rep`. In this case, the following *non-equivalent* rewriting improves the communication speed of an algorithm, while preserving its computational efficiency and overall meaning:

$$\texttt{rep}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \texttt{e}_2[\texttt{nbr}\{\texttt{x}\}]\} \longrightarrow \texttt{share}(\texttt{e}_1)\{(\texttt{x}) \texttt{=>} \texttt{e}_2[\texttt{x}]\}$$

In other words, the `share` construct can be used to *automatically* improve communication speeds of algorithms. Many algorithms with more varied uses of `x` (e.g., using both `x` and $\texttt{nbr}\{\texttt{x}\}$ in $\texttt{e}_2$) can be similarly transformed into improved versions.

3.1 Typing and Operational Semantics

Formal typing and operational semantics for the `share` construct is presented in Figure 3 (bottom frame), as an extension to the type system and semantics given in [5]. The typing judgement $\mathcal{A} \vdash \texttt{e} : T$ is to be read "expression `e` has type T under the set of assumptions $\mathcal{A}$". The typing rule [T-SHARE] requires both $\texttt{e}_1$ and $\texttt{e}_2$ to have a same local (i.e. non-field) type L, assuming `x` to have the

Value-trees and value-tree environments:

$\theta ::= \mathtt{v}\langle\overline{\theta}\rangle$ value-tree

$\Theta ::= \overline{\delta} \mapsto \overline{\theta}$ value-tree environment

Auxiliary functions:

$$\phi_0[\phi_1] = \phi_2 \text{ where } \phi_2(\delta) = \begin{cases} \phi_1(\delta) \text{ if } \delta \in \mathbf{dom}(\phi_1) \\ \phi_0(\delta) \text{ otherwise} \end{cases}$$

$$\rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$$

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

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

Rules for typing and expression evaluation:

$$\text{[T-SHARE]} \quad \frac{\mathcal{A} \vdash \mathtt{e}_1 : L \qquad \mathcal{A}, \mathtt{x} : \mathtt{field}(L) \vdash \mathtt{e}_2 : L}{\mathcal{A} \vdash \mathtt{share}(\mathtt{e}_1)\{(\mathtt{x}) \texttt{ => } \mathtt{e}_2\} : L}$$

$$\text{[E-SHARE]} \quad \frac{\begin{array}{ll} \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 & \phi' = \rho(\pi_2(\Theta)) \\ \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2[\mathtt{x} := \phi] \Downarrow \theta_2 & \phi = (\delta \mapsto \rho(\theta_1))[\phi'] \end{array}}{\delta; \Theta; \sigma \vdash \mathtt{share}(\mathtt{e}_1)\{(\mathtt{x}) \texttt{ => } \mathtt{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}$$

Fig. 3. Typing and operational semantics for the `share` construct.

corresponding field type $\mathtt{field}(L)$, and assigns the same type L to the whole construct.

The evaluation rule is based on the auxiliary functions given in Figure 3 (middle frame). Function $\rho(\theta)$ extracts the root from a given value-tree, while function $\pi_i(\theta)$ selects the i-th sub-tree of the given value-tree. Both of them can be applied to value-tree environments Θ as well, obtaining a neighboring field (for ρ) or another value-tree environment (for π_i). Furthermore, 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 $\mathtt{e}_1$ and $\mathtt{e}_2$ respectively). First, it evaluates $\mathtt{e}_1$ with respect to the corresponding branches of neighbors $\pi_1(\Theta)$ obtaining θ_1. Then, it collects the results for the construct from neighbors into the neighboring field $\phi' = \rho(\pi_2(\Theta))$. 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 $\mathtt{x}$ in the evaluation of $\mathtt{e}_2$ (with respect to the corresponding branches of neighbors $\pi_2(\Theta)$) obtaining θ_2, setting $\rho(\theta_2)$ to be the overall value for the construct.

3.2 The `share` Construct Improves Communication Speed

To illustrate how the `share` construct solves the problem illustrated in Section 2.3, let us once again consider the `ever` function discussed in that section, for propagating when a `condition` Boolean has ever become true. With the `share` construct, we can finally write a fully functional implementation of `ever` as follows:

```
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 does so `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 `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 alone, such optimal communication speed can be achieved only through unbounded recursion, as argued in [2] and reviewed above in Section 2.3.

The average improvement in communication speed of a routine being converted from the usage of `rep`+`nbr` to `share` according to the rewriting proposed at the beginning of this section 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 neighbor: prototypical examples of such algorithms are distance estimations [4], which are computed out of the value of the single neighbor on the optimal path to the source. In this case, letting T be the average interval between subsequent rounds, the 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 neighbors: prototypical examples of such algorithms are data collections [3], especially when they are idempotent (e.g. minimums or maximums). In this case, the existence of a single communication path $\epsilon_0 \rightsquigarrow \ldots \rightsquigarrow \epsilon$ 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 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 ap-

plied this development by extending the Protelis [30] implementation of field calculus to support `share`. We have further upgraded every function in the `protelis-lang` library [15] 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. 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.

4.1 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 $\pm$0.1 Hz, and communicating within a range of 75 meters. All aggregate programs have been written in Protelis [30] and simulations performed using Alchemist [29]. 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 [20] and matplotlib [21]. For the sake of brevity, we do not report the actual code in this paper; however, to guarantee the complete reproducibility of the experiments, the execution of the experiment has been entirely automated, and all the resources have been made publicly available along with instructions.[6]

In the first scenario, we position 2000 mobile devices into a corridor room with sides of, respectively, 200m and 2000m. All but two of the devices are free to move within the corridor randomly, while the remaining two are "sources" that are fixed and located at opposite ends of the corridor. 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 execute the following commonly used coordination algorithms:

1. they compute a potential field measuring the distance from the active source through BIS [4] (`bisGradient` routine in `protelis:coord:spreading`);
2. they accumulate the maximum distance value descending the potential towards the source, through Parametric Weighted Multi-Path C [3] (an optimized version of `C` in `protelis:coord:accumulation`);
3. they broadcast the information along the potential, from the source to every other device in the system (an optimized version of the `broadcast` algorithm found 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 revealed to be 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

[6] `https://bitbucket.org/danysk/experiment-2019-coordination-aggregate-share/`

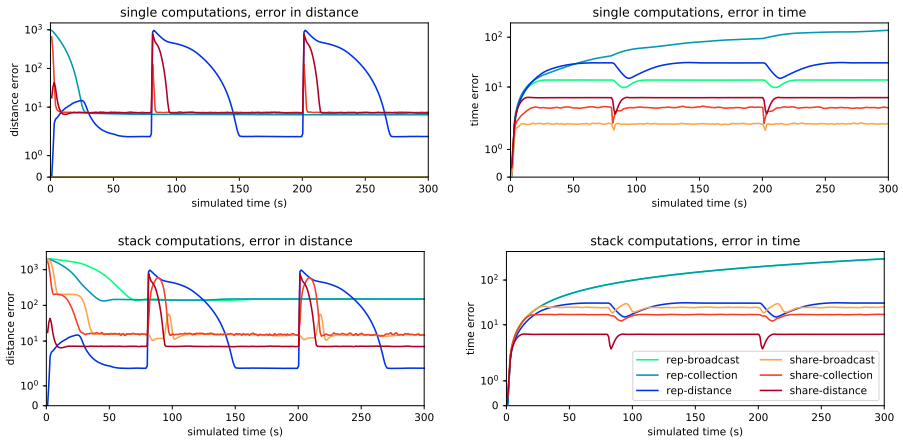

Fig. 4. Performance in the corridor scenario, for both individual algorithms (top) and the composite computation (bottom). Vertical axis is linear in $[0, 1]$ and logarithmic above. Charts on the left column show distance error, while the right column shows time error. The versions of the algorithms implemented with `share` (warm colors) produce significantly less error and converge significantly faster in case of large disruptions than with `rep` (cold colors).

instabilities for more ad-hoc algorithms, by allowing them to propagate quicker and more freely. We thus had to select the most reliable algorithms: optimal single-path distance estimation, optimal multi-path broadcast, and fine tune the latest state-of-the-art version of data collection.

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). 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:

1. composite behavior, in which each step is fed the result computed by the previous step, and
2. individual behavior, in which each step is fed an ideal result for the previous step, as provided by an oracle.

Figure 4 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`.

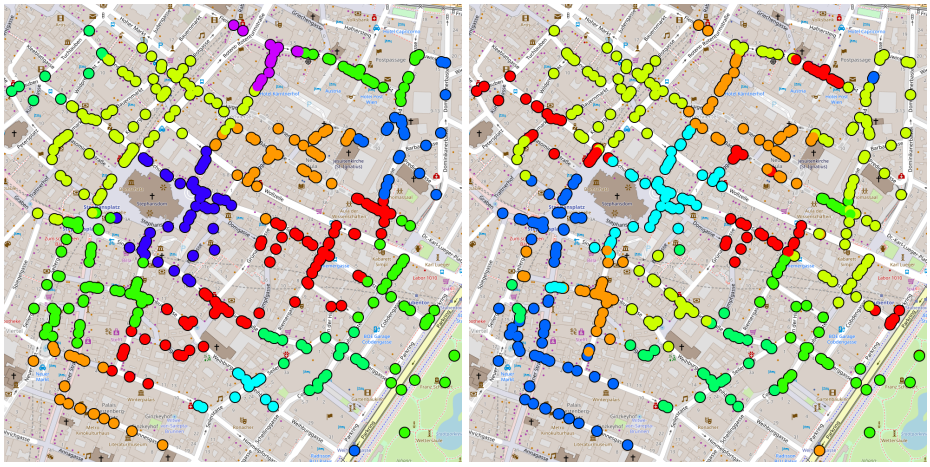

Fig. 5. Snapshots of the Voronoi partitioning scenario using `share` (left) or `rep` (right). Colored dots are simulated devices, with each region having a different color. Faster communication with `share` leads to a higher accuracy distance estimation, allowing the `share` implementation to perform a better division into regions and preventing regions from expanding beyond their limits.

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`.

In the second example, we deploy 500 devices in the city center of a European city, 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 [19]). In this scenario, devices must self-organise service management regions with a radius of at most 200 meters, creating a Voronoi partition as shown in Figure 5 (functions `S` and `voronoiPatitioningWithMetric` 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 $\epsilon_n = \max(0, d(n, l_n) - r)$, 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 6 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

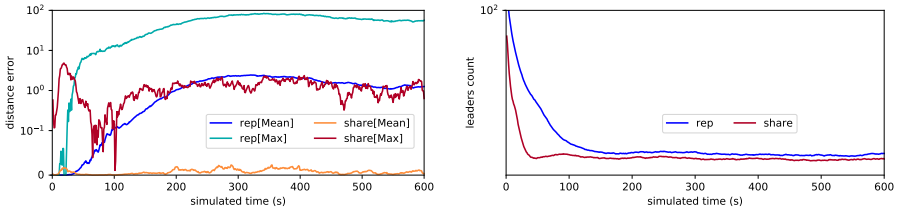

Fig. 6. Performance in the Voronoi partition scenario: error in distance on the left, leaders count with time on the right. Vertical axis is linear in $[0, 0.1]$ and logarithmic elsewhere. The version implemented with `share` has much lower error: the mean error is negligible, and the most incorrect 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.

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.

5 Contributions and Future Work

In this work, we have introduced a novel `share` construct whose introduction allows a significant acceleration of field calculus programs. We have also made this construct available for use in applications though 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.

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 library. Furthermore, we intend to fully analyze the consequences of `share` for improvement of the space-time universality of field calculus. It also appears likely that 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 also anticipate that the improvements in performance will also have positive consequences for nearly all current and future applications that are making use of field calculus.

References

1. Ashley-Rollman, M.P., Goldstein, S.C., Lee, P., Mowry, T.C., Pillai, P.: Meld: A declarative approach to programming ensembles. In: IEEE International Conference on Intelligent Robots and Systems (IROS '07). pp. 2794–2800 (2007)

2. Audrito, G., Beal, J., Damiani, F., Viroli, M.: Space-time universality of field calculus. In: Coordination Models and Languages. Lecture Notes in Computer Science, vol. 10852, pp. 1–20. Springer (2018)
3. Audrito, G., Bergamini, S., Damiani, F., Viroli, M.: Effective collective summarisation of distributed data in mobile multi-agent systems. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS), to appear. ACM (2019)
4. Audrito, G., Damiani, F., Viroli, M.: Optimally-self-healing distributed gradient structures through bounded information speed. In: COORDINATION 2017. LNCS, vol. 10319, pp. 59–77. Springer (2017), best paper at Coordination 2017
5. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Logic 20(1), 5:1–5:55 (2019)
6. Beal, J., Bachrach, J.: Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intelligent Systems 21, 10–19 (March/April 2006)
7. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, chap. 16, pp. 436–501. IGI Global (2013)
8. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer 48(9) (2015)
9. Butera, W.: Programming a Paintable Computer. Ph.D. thesis, MIT, Cambridge, USA (2002)
10. Church, A.: A set of postulates for the foundation of logic. Annals of Mathematics 33(2), 346–366 (1932)
11. Clement, L., Nagpal, R.: Self-assembly and self-repairing topologies. In: Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open (2003)
12. Coore, D.: Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer. Ph.D. thesis, MIT, Cambridge, MA, USA (1999)
13. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Mobile data collection in sensor networks: The tinylime middleware. Elsevier Pervasive and Mobile Computing Journal 4, 446–469 (2005)
14. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
15. Francia, M., Pianini, D., Beal, J., Viroli, M.: Towards a foundational api for resilient distributed systems design. In: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W). pp. 27–32. IEEE (2017)
16. Giavitto, J.L., Godin, C., Michel, O., Prusinkiewicz, P.: Computational models for integrative and developmental biology. Tech. Rep. 72-2002, Univerite d'Evry, LaMI (2002)
17. Giavitto, J.L., Michel, O., Cohen, J., Spicher, A.: Computations in space and space in computations. In: Unconventional Programming Paradigms, Lecture Notes in Computer Science, vol. 3566, pp. 137–152. Springer, Berlin (2005)
18. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using kairos. In: Distributed Computing in Sensor Systems (DCOSS). pp. 126–140 (2005)
19. Haklay, M., Weber, P.: OpenStreetMap: User-generated street maps. IEEE Pervasive Computing 7(4), 12–18 (oct 2008), https://doi.org/10.1109/mprv.2008.80
20. Hoyer, S., Hamman, J.: xarray: N-D labeled arrays and datasets in Python. Journal of Open Research Software 5(1) (2017), http://doi.org/10.5334/jors.148
21. Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing In Science & Engineering 9(3), 90–95 (2007)

22. Kondacs, A.: Biologically-inspired self-assembly of 2d shapes, using global-to-local compilation. In: International Joint Conference on Artificial Intelligence (IJCAI) (2003)
23. Lasser, C., Massar, J., Miney, J., Dayton, L.: Starlisp Reference Manual. Thinking Machines Corporation (1988)
24. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TAG: A Tiny AGgregation Service for Ad-hoc Sensor Networks. SIGOPS Oper. Syst. Rev. 36, 131–146 (2002)
25. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The tota approach. ACM Trans. on Software Engineering Methodologies 18(4), 1–56 (2009)
26. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I. Information and Computation 100(1), 1–40 (September 1992)
27. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. Ph.D. thesis, MIT, Cambridge, MA, USA (2001)
28. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks. pp. 78–87 (aug 2004)
29. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation 7(3), 202–215 (2013)
30. Pianini, D., Viroli, M., Beal, J.: Protelis: Practical aggregate programming. In: ACM Symposium on Applied Computing 2015. pp. 1846–1853 (April 2015)
31. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
32. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. 28(2), 16:1–16:28 (2018)
33. Viroli, M., Pianini, D., Montagna, S., Stevenson, G., Zambonelli, F.: A coordination model of pervasive service ecosystems. Science of Computer Programming 110, 3 – 22 (2015)
34. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proceedings of the 2nd international conference on Mobile systems, applications, and services. ACM Press (2004)
35. Yamins, D.: A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems. Ph.D. thesis, Harvard, Cambridge, MA, USA (December 2007)
36. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Record 31, 2002 (2002)

A Appendix: Field Calculus Operational Semantics

This appendix is not part of the submission,
and it is included for referees' convenience only.

A.1 Device Semantics

The computation that takes place on a single device is formalised by a big-step semantics, expressed by the judgement $\delta; \Theta; \sigma \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta$, to be read "expression $\mathtt{e}_{\mathtt{main}}$ evaluates to θ on device δ with respect to environment Θ and 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 $\mathtt{e}_{\mathtt{main}}$. Such a result is made available to δ's neighbors for their subsequent firing (including δ itself, so as to support a form of state across computation rounds). The recently-received value-trees of neighbors are then collected into a *value-tree environment* Θ, implemented as a map from device identifiers to value-trees (written $\overline{\delta} \mapsto \overline{\theta}$ as short for $\delta_1 \mapsto \theta_1, \ldots, \delta_n \mapsto \theta_n$). Intuitively, the outcome of the evaluation will depend on those value-trees. Figure 7 (top) defines value-trees and value-tree environments.

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

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

Figure 7 (bottom) defines the judgement $\delta; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta$, where: *(i)* δ is the identifier of the current device; *(ii)* Θ is the neighboring field of the value-trees produced by the most recent evaluation of (an expression corresponding to) $\mathtt{e}$ on δ's neighbors; *(iii)* $\mathtt{e}$ is a closed run-time expression (i.e., a closed expression that may contain neighboring field values); *(iv)* the value-tree θ represents the values computed for all the expressions encountered during the evaluation of $\mathtt{e}$—in particular the root of the value tree θ, denoted by $\rho(\theta)$, is the value computed for expression $\mathtt{e}$. The auxiliary function ρ is defined in Figure 7 (second frame).

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression $\mathtt{e}'$ of $\mathtt{e}$ with respect to the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ. This process, called *alignment*, is modelled by the auxiliary function π defined in Figure 7 (second frame). This function has two different behaviors (specified

Value-trees and value-tree environments:

$\theta ::= \mathtt{v}\langle\overline{\theta}\rangle$ value-tree

$\Theta ::= \overline{\delta} \mapsto \overline{\theta}$ value-tree environment

Auxiliary functions:

$\rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$

$\pi_i(\mathtt{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_i$ if $1 \le i \le n$ $\qquad$ $\pi^\ell(\mathtt{v}\langle\theta_1,\theta_2\rangle) = \theta_2$ if $\rho(\theta_1) = \ell$

$\pi_i(\theta) = \bullet$ otherwise $\qquad$ $\pi^\ell(\theta) = \bullet$ otherwise

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

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

Syntactic shorthands:

$\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta}$ where $|\overline{\mathtt{e}}| = n$ for $\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \cdots \delta; \pi_n(\Theta); \sigma \vdash \mathtt{e}_n \Downarrow \theta_n$

$\rho(\overline{\theta})$ where $|\overline{\theta}| = n$ for $\rho(\theta_1), \ldots, \rho(\theta_n)$

$\overline{\mathtt{x}} := \rho(\overline{\theta})$ where $|\overline{\mathtt{x}}| = n$ for $\mathtt{x}_1 := \rho(\theta_1)\ \ldots\ \mathtt{x}_n := \rho(\theta_n)$

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

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

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

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

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

$$\frac{\begin{array}{ll}\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 & \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \rho(\theta_1) & \text{otherwise} \end{cases} \\ \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2[\mathtt{x} := \ell_0] \Downarrow \theta_2 & \end{array}}{\delta; \Theta; \sigma \vdash \mathtt{rep(e_1)\{(x)\ =>\ e_2\}} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}\ [\text{E-REP}]$$

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

Fig. 7. Big-step operational semantics for expression evaluation.

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 7 (second frame).

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

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $\mathtt{b}(\mathtt{e}_1 \cdots \mathtt{e}_n)$, where $n \geq 0$. It produces the value-tree $\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle$, where $\theta_1, \ldots, \theta_n$ are the value-trees produced by the evaluation of the actual parameters $\mathtt{e}_1, \ldots, \mathtt{e}_n$ and $\mathtt{v}$ is the value returned by the function. The rule exploits the special auxiliary function $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$, whose actual definition is abstracted away. This is such that $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}(\overline{\mathtt{v}})$ computes the result of applying built-in function $\mathtt{b}$ to values $\overline{\mathtt{v}}$ in the current environment of the device δ. In particular: the built-in 0-ary function `self` gets evaluated to the current device identifier (i.e., $(\!|\mathtt{self}|\!)^{\Theta}_{\delta}() = \delta$), and mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $(\!|+|\!)^{\Theta}_{\delta}(2,3) = 5$).

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

The $(\!|\mathtt{b}|\!)^{\Theta}_{\delta}$ 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 $\mathtt{d}(\mathtt{e}_1 \ldots \mathtt{e}_n)$, 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 $\mathtt{d}$ (denoted by $\mathit{body}(\mathtt{d})$) substituting the formal parameters of the function (denoted by $\mathit{args}(\mathtt{d})$) with the values obtained evaluating $\mathtt{e}_1, \ldots \mathtt{e}_n$.

Rule [E-REP] implements internal state evolution through computational rounds: $\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x})\ \mathtt{=>}\ \mathtt{e}_2\}$ evaluates to $\mathtt{e}_2[\mathtt{x} := \mathtt{v}]$ where $\mathtt{v}$ is obtained from $\mathtt{e}_1$ on the first firing of a device, from the previous value of the whole `rep`-expression otherwise.

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

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

Example 4. To illustrate rule [E-NBR], consider $\mathtt{e}' = \mathtt{minHood}(\mathtt{nbr}\{\mathtt{snsNum}()\})$, where the 1-ary built-in function `minHood` returns the lower limit of values in the range of its neighboring field argument, and the 0-ary built-in function `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;
- `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 $\mathtt{snsNum}()$ on δ_A yields 1 (by rules [E-LOC] and [E-B-APP], since $(\!|\mathtt{snsNum}|\!)^{\emptyset}_{\delta_A}() = 1$); the evaluation of $\mathtt{nbr}\{\mathtt{snsNum}()\}$ on δ_A yields $(\delta_A \mapsto 1)\langle 2\rangle$ (by rule [E-NBR]); and the evaluation of $\mathtt{e}'$ on δ_A yields

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

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

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

Suppose that device δ_B is the third device that fires. Then the evaluation of $\mathtt{e}'$ 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 $\mathtt{nbr}\{\mathtt{snsNum}()\}$ and $\mathtt{snsNum}()$ is performed, respectively, with respect to the following value-tree environments obtained from Θ_B by alignment:

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

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

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

System configurations and action labels:

Ψ	$::= \overline{\delta} \mapsto \overline{\Theta}$	status field
τ	$::= \overline{\delta} \mapsto \overline{I}$	topology
Σ	$::= \overline{\delta} \mapsto \overline{\sigma}$	sensors-map
Env	$::= \tau, \Sigma$	environment
N	$::= \langle Env; \Psi \rangle$	network configuration
act	$::= \delta \mid env$	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: $\boxed{N \xrightarrow{act} N}$

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

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

Fig. 8. Small-step operational semantics for network evolution.

A.2 Network Semantics

The overall network evolution is formalised by the small-step operational semantics given in Figure 8 as a transition system on network configurations N. Figure 8 (top) defines key syntactic elements to this end. Ψ models the overall status of the devices in the network at a given time, as a map from device identifiers to value-tree environments. From it, we can define the state of the field at that time by summarising the current values held by devices. τ models *network topology*, namely, a directed neighboring 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. So, a whole network configuration N is a couple of a status field and environment.

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

We consider transitions $N \xrightarrow{act} N'$ of two kinds: firings, where act is the corresponding device identifier, and environment changes, where act is the special label env. This is formalised in Figure 8 (bottom). Rule [N-FIR] models a computation round (firing) at device δ: it takes the local value-tree environment

filtered out of old values $F(\Psi)(\delta)$;[7] then by the single device semantics it obtains the device's value-tree θ,[8] which is used to update the system configuration of δ and of δ's neighbors.

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 8 (middle). Let $\overline{\delta}$ be the domain of Env'. We first construct a status field Ψ_0 associating to all the devices of Env' the empty context $\emptyset$. Then, we adapt the existing status field Ψ to the new set of devices: $\Psi_0[\Psi]$ automatically handles removal of devices, map of new devices to the empty context, and retention of existing contexts in the other devices.

Example 5. Consider a network of devices with $\mathtt{e}' = \mathtt{minHood}(\mathtt{nbr}\{\mathtt{snsNum}()\})$ as introduced in Example 4. The network configuration illustrated at the beginning of Example 4 can be generated by applying rule [N-ENV] to the empty network configuration. I.e., we have

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

where

- $Env_0 = \tau_0, \Sigma_0$,
- $\tau_0 = (\delta_A \mapsto \{\delta_B\}, \delta_B \mapsto \{\delta_A, \delta_C\}, \delta_C \mapsto \{\delta_B\})$,
- $\Sigma_0 = (\delta_A \mapsto (\mathtt{snsNum} \mapsto 1), \delta_B \mapsto (\mathtt{snsNum} \mapsto 2), \delta_C \mapsto (\mathtt{snsNum} \mapsto 3))$, and
- $\Psi_0 = (\delta_A \mapsto \emptyset, \delta_B \mapsto \emptyset, \delta_C \mapsto \emptyset)$.

Then, the tree fires of devices δ_A, δ_C and δ_B illustrated in Example 4 correspond to the following transitions, respectively.

1. $\langle Env_0; \Psi_0 \rangle \xrightarrow{\delta_A} \langle Env_0; \Psi' \rangle$, where
   - $\Psi' = (\delta_A \mapsto (\delta_A \mapsto \theta_A),\ \delta_B \mapsto (\delta_A \mapsto \theta_A),\ \delta_C \mapsto \emptyset)$, and
   - $\theta_A = 1\langle(\delta_A \mapsto 1)\langle 1 \rangle\rangle$;
2. $\langle Env_0; \Psi' \rangle \xrightarrow{\delta_C} \langle Env_0; \Psi'' \rangle$, where
   - $\Psi'' = (\delta_A \mapsto (\delta_A \mapsto \theta_A),\ \delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_C \mapsto \theta_C),\ \delta_C \mapsto (\delta_C \mapsto \theta_C))$, and
   - $\theta_C = 1\langle(\delta_C \mapsto 3)\langle 3 \rangle\rangle$;
3. $\langle Env_0; \Psi'' \rangle \xrightarrow{\delta_B} \langle Env_0; \Psi''' \rangle$, where
   - $\Psi''' = (\delta_A \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B),$
     $\delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C),$
     $\delta_C \mapsto (\delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C))$,
   - $\theta_B = 1\langle\phi\langle 2 \rangle\rangle$, and
   - $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$.

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

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