# Functional programming for distributed systems with XC

**Giorgio Audrito**
University of Turin, Turin, Italy

**Roberto Casadei**
University of Bologna, Cesena, Italy

**Ferruccio Damiani**
University of Turin, Turin, Italy

**Guido Salvaneschi**
University of St.Gallen, St.Gallen, Switzerland

**Mirko Viroli**
University of Bologna, Cesena, Italy

## Abstract

Programming distributed systems is notoriously hard due to – among the others – concurrency, asynchronous execution, message loss, and device failures. *Homogeneous* distributed systems consist of similar devices that communicate to neighbours and execute the same program: they include wireless sensor networks, network hardware and robot swarms. For the homogeneous case, we investigate an experimental language design that aims to push the abstraction boundaries farther, compared to existing approaches.

In this paper, we introduce the design of XC, a programming language to develop homogeneous distributed systems. In XC, developers define the single program that every device executes and the overall behaviour is achieved collectively, in an emergent way. The programming framework abstracts over concurrency, asynchronous execution, message loss, and device failures. We propose a minimalistic design, which features a single declarative primitive for communication, state management and connection management. A mechanism called *alignment* enables developers to abstract over asynchronous execution while still retaining composability. We define syntax and operational semantics of a core calculus, and briefly discuss its main properties. XC comes with two DSL implementations: a DSL in Scala and one in C++. An evaluation based on smart-city monitoring demonstrates XC in a realistic application.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models; Theory of computation → Functional constructs; Theory of computation → Operational semantics; Theory of computation → Type structures; Computing methodologies → Distributed programming languages

**Keywords and phrases** Core calculus, operational semantics, type soundness, Scala DSL

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2022.20

**Acknowledgements** This work was supported by the EU/MUR FSE REACT-EU PON R&I 2014-2022 (CCI2014IT16M2OP005), the Swiss National Science Foundation (SNSF, No. 200429), the Hessian LOEWE initiative emergenCITY, and the Ateneo/CSP "Bando ex post 2020".

## 1 Introduction

Programming distributed systems is notoriously hard because they require reasoning about a number of issues that inevitably arise in this setting, including concurrency, remote communication, asynchronous execution, message loss and device failures.

The design of programming languages for distributed systems attempts to address these issues by carefully combining cases where *(i)* programmers are given *explicit control* over certain aspects of distribution, *(ii)* the design employs *abstraction* to hide low level mechanisms

© Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi and Mirko Viroli;
licensed under Creative Commons License CC-BY 4.0

from developers. A well-known example of explicit control is fault tolerance in the actor model, where developers can define a reaction strategy in case of failures (through the so-called actor supervision hierarchy) [3]. However, the actor model abstracts over placement and – at least in the original actor model – actors communicate with actors on the same machine the same way they do with a remote actor. Similar combinations of abstraction and explicit control position other distributed programming languages in the design space. For example, in the MPI model for HPC, processes are organized in topologies and can explicitly send messages close to them in such topology [45]. In Partitioned Global Address Space Languages [28], the programming model abstracts over the memory separation across processes. Pub-sub systems abstract over the binding between message sender and receivers ensuring that senders and receivers can seamlessly join and leave the system [31]. In summary, the design of these programming models stems from a combination of explicit control ensured to the user and details that are abstracted over.

An important class of distributed systems are *homogeneous* and *situated*, i.e., they are composed of similar devices that communicate with 'neighbours', and execute similar programs. This property has been observed, e.g., in distributed systems for hierarchical control of network routing [37], crowd management by handheld devices [16], Wireless Sensor Network (WSN) connectivity management [36] and gossip-based data aggregation [35], task allocation in robot swarms [18, 46], and coordination of enterprise servers [24]. More applications are emerging, pushed by the scientific and technological trends of the Internet of Things (IoT) and of Cyber-Physical Systems (CPS) [47], and of the coordination of mobile agents [40]. Crucially, 'homogeneity' in large-scale systems also stems from the case where each device runs a program from a predefined set – corresponding to a homogeneous configuration with a single program with an initial branch.

In this paper, we address the issue of programming such a class of homogeneous systems. Over time, several approaches have been proposed to address these kinds of systems, including *spatial computing* [29], *ensemble-based programming* [27, 1], and, notably, *field-based computing* [49, 38, 40], where the overall distributed system behaviour is understood as producing a *computational field*, i.e., a map from network nodes to values. Inspired by these works, we investigate XC, a novel programming language design that captures their essence (as detailed in Section 7) *into a single construct* aiming to abstract over low-level concerns developers face in distributed systems, while allowing differentiated messages to neighbours. We show a design where concurrency, asynchronicity, network communication, message loss, and failures do not need to be handled explicitly. Thanks to a mechanism that is referred to as *alignment*, distributed programs written in this style retain composability even if devices operate fully independently, waking up, executing the program and communicating asynchronously at arbitrary times and frequencies. Messages from other devices are processed homogeneously, hence developers do not need to separately handle the case when a message is lost or a device fails: such lost message is simply not part of the (homogeneous) computation. All required computational mechanisms can be unified into a single declarative construct called `exchange`, which provides *(i)* access to neighbours' values, *(ii)* persistence of information for subsequent executions, *(iii)* communication with neighbours, and *(iv)* compositional behaviour.

To summarise, this paper provides the following contributions:

- We describe the design of XC, a programming language for homogeneous distributed systems that abstracts over concurrency, network communication, message loss, and device failures. Crucially, XC retains compositionality even with asynchronous communication, thanks to alignment.
- We show that XC can effectively capture a number of applications in distributed systems,

including distributed protocols such as gossiping, finding an optimised communication channel, and common applications in self-organizing systems.
- We provide a formalization of a core calculus for XC, including syntax and operational semantics, and briefly discuss its main properties.
- We implement XC as publicly available Scala and C++ internal DSLs, together targeting a number of different execution platforms.
- In addition to the applications above, we evaluate our approach on a case study demonstrating XC's applicability to real-world scenarios and its compositionality, and answering two research questions: *(RQ1)* whether the decentralised execution of the XC program on each device induces the desired *collective* behaviour; and *(RQ2)* to what extent such behaviour can be expressed by composition of simpler functions.

The paper is structured as follows. Section 2 introduces XC design, Section 3 demonstrates XC through examples, Section 4 presents the formalization, Section 5 discusses the implementation, Section 6 evaluates XC, Section 7 compares XC to the related work, Section 8 concludes and outlines future research directions.

# 2 XC Language Design

## 2.1 System model

**Asynchronous, round-based execution and communication.** We consider *devices* that send/receive *messages* with (physical or logical) *neighbours*. The set of neighbours of any device can change dynamically to model, e.g., spatial movement, failures, and network delay. Existing homogeneous systems (cf. [15]) typically work with devices that repeatedly execute a computation aimed at producing a message for some neighbours, whereas message reception is asynchronous. Therefore, we abstract device behaviour through a notion of *(execution) round*, whereby a device independently "fires" and "atomically executes" a XC program, then it sends a resulting message to neighbours before waiting to execute the round again—sometimes we say it "wakes up", execute the round, and then "go back to sleep". The behaviour of each device in the network is developed as a single program[1]. Such execution rounds may occur at comparable periodic intervals on all devices but there is no such assumption in general (every device may have its own scheduling of rounds). Indeed, a device can run out of battery and never wake up again, or it can restart waking up after a long time if the battery gets charged. In summary, rounds—hence the communication among devices—are entirely asynchronous.

**Last-message buffering and dropping.** The messages received by a sleeping device queue up in a buffer. When the device wakes up, it executes a XC program that processes such messages, producing new messages to send out. Such messages are eventually processed by the neighbours when they wake up for their next round. For example, in the system execution in Figure 1, there are four devices $\delta_1$ to $\delta_4$. In the considered time span, device $\delta_2$ wakes up twice and performs two computation rounds, $\epsilon_1$ and $\epsilon_2$. Grey arrows indicate messages that get lost and are never received. The computation $\epsilon_2$ processes three messages, received from $\delta_4$, $\delta_3$, and $\delta_1$ while $\delta_2$ was asleep. After the computation, $\delta_2$ sends out a message to $\delta_3$ and to $\delta_1$. The order of messages from a same sender is preserved but, other than that, there are

[1] This approach is often referred to as *macroprogramming* [44] or *multi-tier programming* [50]. It does not restrict realisable behaviours as devices can still exhibit different executions of the same program.

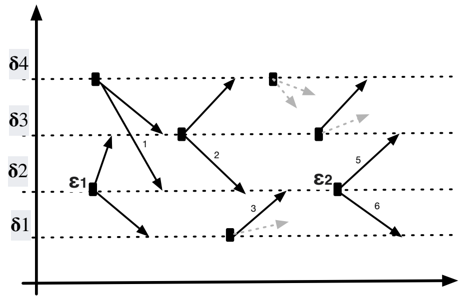



**Figure 1** XC system model

very few assumptions on messages. If a device $\delta_1$ runs multiple rounds before a device $\delta_2$ even runs a single round, $\delta_2$ sees only the message received from the last round of $\delta_1$, i.e., newly received messages from a same sender overwrite older ones. Also, messages are not removed from the buffer after reading them, unless they *expire* (i.e., are deemed too old according to any pre-established criterion) or unless they are replaced by a new message from the same device, allowing messages to (possibly) persist across rounds. The XC design abstracts over the specific expiration criteria: common choices include removing messages after each read, or after a validity time elapses. This time interval is highly application-specific and stems from a trade-off between *(i)* tolerance to communication delays and failures, and *(ii)* recovery speed after truthful changes on data and neighbourhoods.

When a device $\delta$ wakes up, it usually does not find messages from every other device in the system: *(i)* another device may be too far to send a message to $\delta$; *(ii)* messages may get lost; *(iii)* devices may disappear or fail; *(iv)* a device may reboot, losing its queue of received messages; *(v)* $\delta$ may deem messages from some devices to be expired. Crucially, XC does not require distinguishing among those cases. When a device wakes up, it finds some messages from (the most recent available execution round of) some other devices. The devices for which a message is available in a certain round are *the neighbours* for that round.

This system model and the terminology associated to it (e.g., 'send message to a neighbour') is adopted throughout the paper. These design choices make XC agnostic to the actual communication channel, topology creation and discovery mechanism: e.g., push or pull, broadcast or point-to-point. For example, the same programming model would apply even if a device, after waking up, contacts the neighbours to fetch their current value in a *pull* fashion. Instead, in a network of micro-controller devices, Bluetooth 5.0 *extended advertisements* could be used to share data with neighbour devices in physical proximity, without an explicit discovery mechanism, as the topology is induced by the messages that are actually received. Such an implementation would also grant *causal consistency* [2]. On the other hand, a network of higher-end devices may communicate point-to-point over IP, with discovery mechanisms based on broadcasted messages or rendezvous servers.

## 2.2 XC's key data type: Neighbouring Values

**Datatypes in XC.** XC features two kinds of values. *Local* values $\ell$ include traditional types $A$ like float, string or list. Neighbouring values (*nvalues*) are a map $\underline{\mathtt{w}}$ from device identifiers $\delta_i$ to corresponding local values $\ell_i$, with a *default* $\ell$, written $\ell[\delta_1 \mapsto \ell_1, \dots, \delta_n \mapsto \ell_n]$. A nvalue is used to describe the (set of) values received from and sent to neighbours. In highly decoupled distributed systems *only a few* neighbours may *occasionally* produce a value. The devices with an associated entry in the nvalue are hence usually a subset of all devices, e.g., because a device is too far to provide a value or the last provided value has expired. The default is used when a value is not available for some reason as will be discussed later (e.g., if a device just appeared and has not yet produced a value). For this reason, it is convenient to adopt the notation above and read it "the nvalue $\underline{\mathtt{w}}$ is $\ell$ everywhere (i.e., for all neighbours) except for devices $\delta_1, \dots, \delta_n$ with values $\ell_1, \dots, \ell_n$.

To exemplify nvalues, in Figure 1, upon waking up for computation $\epsilon_2$, $\delta_2$ may process a nvalue $\underline{\mathtt{w}} = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2, \delta_1 \mapsto 3]$, corresponding to the messages carrying the scalar values 1, 2, and 3 received when asleep from $\delta_4$, $\delta_3$, and $\delta_1$. The entries for all other devices default to 0. After the computation, $\delta_2$ may send out the messages represented by the nvalue $\underline{\mathtt{w}}' = 0[\delta_3 \mapsto 5, \delta_1 \mapsto 6]$; so that 5 is sent to $\delta_3$, 6 is sent to $\delta_1$, and 0 is sent to every other device (such as a newly-connected device with no dedicated value yet). We also use the notation $\underline{\mathtt{w}}(\delta')$ for the local value $\ell'$ if $\delta' \mapsto \ell'$ is in $\underline{\mathtt{w}}$, or the default local value $\ell$ of $\underline{\mathtt{w}}$ otherwise, reflecting the interpretation of nvalues as maps with a default. For instance, $\underline{\mathtt{w}}'(\delta_1) = 6$ and $\underline{\mathtt{w}}'(\delta_2) = 0$. To help the reader, in code snippets, we underline the variables holding neighbouring values, and, similarly, we underline a primitive type $\underline{A}$ to indicate the type of a nvalue $\underline{\mathtt{w}} = \ell[\delta_1 \mapsto \ell_1, \dots, \delta_n \mapsto \ell_n]$ where $\ell, \ell_1, \dots \ell_n$ have type $A$.

**Nvalues generalize local values.** A local value $\ell$ can be automatically converted to a nvalue $\ell[]$ with a default value for every device. In fact, the distinction between local values and nvalues is only for clarity: local values can be considered equivalent to nvalues where all devices are mapped to a default value. In the formalization (Section 4) local values and nvalues are treated uniformly. Functions on local values are implicitly lifted to nvalues, by applying them on the maps' content pointwise. For example, given $\underline{\mathtt{w}}_1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2]$ and $\underline{\mathtt{w}}_2 = 1[\delta_4 \mapsto 2]$, we have $\underline{\mathtt{w}}_3 = \underline{\mathtt{w}}_1 + \underline{\mathtt{w}}_2 = 1[\delta_4 \mapsto 3, \delta_3 \mapsto 3]$. Note that $\delta_3 \mapsto 3$ in $\underline{\mathtt{w}}_3$ is due to the fact that $\delta_3 \mapsto 2$ in $\underline{\mathtt{w}}_1$ and $\delta_3$ has default value 1 in $\underline{\mathtt{w}}_2$. Using also the automatic promotion of local values to nvalues, we have that $\underline{\mathtt{w}}_1 + 1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2] + 1 = 1[\delta_4 \mapsto 2, \delta_3 \mapsto 3]$.

**Operations on nvalues.** Besides pointwise manipulation, nvalues can be folded over, similar to a list, through built-in function $\mathtt{nfold}(f : (A, B) \to A, \underline{\mathtt{w}} : \underline{B}, \ell : A) : A$, where the function $f$ is repeatedly applied to neighbours' values in a field $\underline{\mathtt{w}}$ (thus excluding the value for the *self* device), starting from a base local value $\ell$. For instance, assume that $\delta_2$ is performing a `nfold` operation, with the current set of neighbours $\{\delta_1, \delta_3\}$. Then $\mathtt{nfold}(+, \underline{\mathtt{w}}_1, 10) = 10 + \underline{\mathtt{w}}_1(\delta_1) + \underline{\mathtt{w}}_1(\delta_3) = 10 + 0 + 2$, where $\underline{\mathtt{w}}_1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2]$ is as above. As nvalues should be agnostic to the ordering of the elements (i.e., the ordering of the identifiers $\delta'$), we usually assume that $f$ is associative and commutative.

**Sensors and actuators.** Since XC programs may express the collective behaviour of homogeneous systems situated in some (physical or computational) environment, the devices are typically equipped with *sensors* and *actuators*. Sensors, in particular, are meant to provide access to contextual and environmental information. These can be accessed by the program through built-in functions as shown in next sections. In a round, similarly to how messages are considered, the program is executed against the most recent sample of sensor values. On the other hand, actuators can be run at the end of the round against the program output (which may collect all the desired actuation commands).

▶ **Example 1** (Distance estimation)**.** A node can estimate its distance from another node in the network by leveraging an existing estimate $\underline{\mathtt{n}}$ provided by its neighbours. To this end, one selects the minimum (using `nfold` with starting value `Infinity`) of neighbours' estimates $\underline{\mathtt{n}}$ increased by the relative distance estimates $\underline{\mathtt{senseDist}}$ (provided by a sensor in the device).

| Name | Type scheme | Description |
|---|---|---|
| **Communication:** | | |
| exchange | $(A, (\underline{A}) \to (T, \underline{A})) \to T$ | *Exchanges messages* |
| **Neighbouring value manipulation:** | | |
| nfold | $((A, B) \to A, \underline{B}, A) \to A$ | *Folding of a neighbouring value* |
| self | $(\underline{A}) \to A$ | *Extract the self-message* |
| updateSelf | $(\underline{A}, A) \to \underline{A}$ | *Update the self-message* |
| **Sensors used in examples:** | | |
| uid | num | *Unique device identifier* |
| senseDist | $\underline{\texttt{num}}$ | *Distance estimates to neighbours* |
| **Point-wise operators:** | | |
| +, -, *, / | (num, num) → num | *Arithmetic operators* |
| and, or | (bool, bool) → bool | *Boolean operators* |
| ==, <=, >= | $(A, A) \to$ bool | *Relational operators*[2] |
| mux | $(\texttt{bool}, A, A) \to A$ | *Multiplexer operator* |
| pair | $(A, B) \to (A, B)$ | *Pair creation* |
| fst | $((A, B)) \to A$ | *First element of a pair* |
| snd | $((A, B)) \to B$ | *Second element of a pair* |
| **Constructors:** | | |
| −1, 0, 0.25, 1, Infinity | num | *Numeric constructors* |
| True, False | bool | *Boolean constructors* |
| Pair | $(A, B) \to (A, B)$ | *Pair constructor* |

**Figure 2** XC: name, type scheme and description of built-in data constructors and functions.

```
def distanceEstimate(n) { // has type scheme: (num) → num
  nfold(min, n + senseDist, Infinity)
}
```

Notice that $\underline{\texttt{n}}$ and $\underline{\texttt{senseDist}}$ sum up neighbour-wise; if neighbour $\delta$ shares estimate $\underline{\texttt{n}}(\delta)$, the node's best estimate from that neighbour is $\underline{\texttt{n}}(\delta) + \underline{\texttt{senseDist}}(\delta)$. The minimum among all estimates is selected, or `Infinity` if no neighbour is available. ◀

Additional built-in operations on nvalues are $\texttt{self}(\underline{\texttt{w}} : \underline{A}) : A$, which returns the local value $\underline{\texttt{w}}(\delta)$ in $\underline{\texttt{w}}$ for the self device $\delta$, and $\texttt{updateSelf}(\underline{\texttt{w}} : \underline{A}, \ell : A) : \underline{A}$ which returns a nvalue equal to $\underline{\texttt{w}}$ except for the self device $\delta$ – updated to $\ell$. The *substitution* notation stand for defaulted map updates, so that $\texttt{updateSelf}(\underline{\texttt{w}}, \ell) = \underline{\texttt{w}}[\delta \mapsto \ell]$. Indeed, the notation $\ell[\delta_1 \mapsto \ell_1, ...]$ for nvalues can be understood as a substitution updating $\ell$ (the map equal to $\ell$ everywhere) by associating $\ell_n$ to $\delta_n$.

XC operators on nvalues behave uniformly on neighbours to encourage uniform behaviour on each element of a nvalue. This approach is idiomatic in XC and results in a more resilient behaviour – inherently tolerate changes of neighbourhoods between rounds. Yet, non-uniform behaviour can be encoded via built-in function `uid` (combined with communication primitives, Section 2.3), which provides the unique identifier $\delta$ of the current device.

Figure 2 shows a summary of every built-in function used in this paper. Constructors and point-wise operators are standard; the *multiplexer* operator $\texttt{mux}(\ell_1, \ell_2, \ell_3)$ returns $\ell_2$ if $\ell_1$ is `True`, $\ell_3$ otherwise. We also omit `pair` and use the shortcut $(\texttt{v}_1, \texttt{v}_2)$ for pair construction, and use infix notation for binary operators whenever convenient. Built-ins for neighbouring values has just been discussed. We introduce the `exchange` operator in the next section.

## 2.3 Communication in XC: Exchange

XC features a single communication primitive $\texttt{exchange}(\texttt{e}_i, (\underline{\texttt{n}})\texttt{=>}\,\texttt{return}\ \texttt{e}_r\ \texttt{send}\ \texttt{e}_s)$ which de-sugars to $\texttt{exchange}(\texttt{e}_i, (\underline{\texttt{n}})\texttt{=>}(\texttt{e}_r, \texttt{e}_s))$ and is evaluated as follows. *(i)* the device computes

[2] The generic $A$ type in relation-based operators is not allowed to be a function type.

the local value $\ell_i$ of $\mathtt{e}_i$ (the *initial* value). *(ii)* it substitutes variable $\underline{\mathtt{n}}$ with the nvalue $\underline{\mathtt{w}}$ of messages received from the neighbours for this exchange, using $\ell_i$ as default. The exchange returns the (neighbouring or local) value $\mathtt{v}_r$ from the evaluation of $\mathtt{e}_r$. *(iii)* $\mathtt{e}_s$ evaluates to a nvalue $\underline{\mathtt{w}}_s$ consisting of local values to be sent to neighbour devices $\delta'$, that will use their corresponding $\underline{\mathtt{w}}_s(\delta')$ as soon as they wake up and perform their next execution round.

Often, expressions $\mathtt{e}_r$ and $\mathtt{e}_s$ coincide, hence we provide $\mathtt{exchange}(\mathtt{e}_i, (\underline{\mathtt{n}})\, \mathtt{=>}\, \mathtt{retsend}\, \mathtt{e})$ as a shorthand for $\mathtt{exchange}(\mathtt{e}_i, (\underline{\mathtt{n}})\, \mathtt{=>}\, (\mathtt{e}, \mathtt{e}))$. Another common pattern is to access neighbours' values, which we support via $\mathtt{nbr}(\mathtt{e}_i, \mathtt{e}_s) = \mathtt{exchange}(\mathtt{e}_i, (\underline{\mathtt{n}})\, \mathtt{=>}\, \mathtt{return}\ \underline{\mathtt{n}}\ \mathtt{send}\ \mathtt{e}_s)$. In $\mathtt{nbr}(\mathtt{e}_i, \mathtt{e}_s)$, the value of expression $\mathtt{e}_s$ is sent to neighbours, and the values received from them (gathered in $\underline{\mathtt{n}}$ together with the default from $\mathtt{e}_i$) are returned as a nvalue, thus providing a view on neighbours' values of $\mathtt{e}_s$.

It is crucial for the expressivity of XC that *exchange* (hence *nbr*) can send a different value to each neighbour, to allow custom interaction, as exemplified below. Next, we show the *self-organising distance* algorithm which showcases the interplay of *exchange* and *nfold*.

▶ **Example 2** (Ping-pong counter)**.** The following function produces a neighbouring value of "connection counters" with neighbours, i.e., it associates every neighbour to the number of times a mutual connection has been established with it.

```
def ping-pong() { // has type scheme: () → num
  exchange( 0, (n) => retsend n + 1 )
}
```

Every time a device evaluates `ping-pong`, it first gathers a neighbouring value $\underline{\mathtt{w}}$ associating neighbours to their respective connection counter – 0 is for newly connected devices. Expression $\underline{\mathtt{n}} + 1$ is computed substituting $\underline{\mathtt{w}}$ for $\underline{\mathtt{n}}$, incrementing each such counter (including for newly connected devices, which now map to 1). The resulting value $\underline{\mathtt{w}} + 1$ is both returned by the expression and shared with neighbours. As long as a connection between two devices is maintained, each receives a connection counter from the other and increments it before sending it back – overall counting the messages bouncing back-and-forth. Once a connection between devices breaks and the corresponding messages expire, the connection counter resets to 0, then starts increasing again in case a connection is re-established. Crucially, the program sends different values to neighbours to keep a distinct connection counter with each. ◀

▶ **Example 3** (Self-organising distance)**.** Computing the minimum distance from any device of the network to a set of *source* devices results in a *gradient* structure [10]. Gradients are a key self-organisation pattern with several applications like estimating long-range distances and providing directions to move data along minimal paths. Function `distanceTo` offers a simple implementation, consisting of a distributed version of the Bellman-Ford algorithm [26].

```
def distanceTo(src) { // has type scheme: (bool) → num
  exchange( Infinity, (n) => retsend mux(src, 0, distanceEstimate(n)) )
}
```

Its repeated application in a (possibly mobile) network of devices stabilises to the expected distances from devices where `src` is true. The `exchange` expression in the body updates a local estimate of the distance by *(i)* using `Infinity` as default distance; *(ii)* returning distance zero on source devices; *(iii)* in other devices, selecting the minimum of neighbours' estimates increased by the relative distance estimates (Example 1). If such estimated distance is $d$, then $d$ is both shared with neighbours (as a constant map with the same estimate $d$ for every neighbour) and returned by the function. Operator `mux` (i.e., a strict version of `if` that computes both its branches, and then selects the output of one of them as result based on the condition) is needed, as sources, though returning 0, must also evaluate

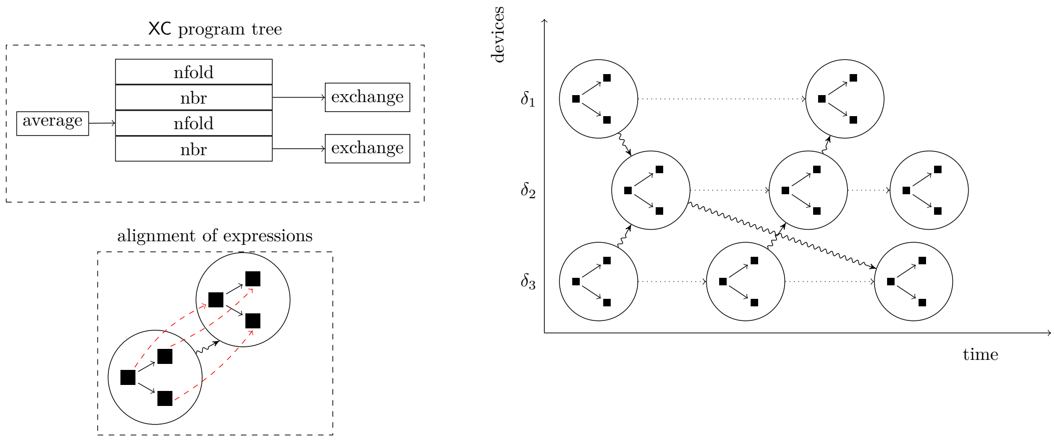



**Figure 3** XC alignment mechanism for Example 4.

function call `distanceEstimate` (thus sharing their value $\underline{\mathtt{n}}$). Any change in the network (e.g., due to failure, mobility, dynamic joining) directly affects the domain of $\underline{\mathtt{n}}$, hence the local computation and eventually the whole network—resulting in inherent adaptiveness. ◀

## 2.4 Compositionality through alignment

If a program executes multiple exchange-expressions, XC ensures through *alignment* that the messages are dispatched to corresponding exchange-expressions across rounds.

▶ **Example 4** (Neighbour average). The following function `average` computes the weighted average of a value across the immediate neighbours of the current device:

```
1 def average(weight, value) { // has type scheme: (num) → num
2   val totW = nfold(+, nbr(0, weight), weight);
3   val totVl = nfold(+, nbr(0, weight*value), weight*value);
4   totV / totW
5 }
```

First, the total weight of neighbours is computed in Line 2, by first producing a nvalue of neighbours' weights through `nbr(0, weight)`, and then reducing it to its total by `nfold`, using `weight` as base value to ensure that the weight of the current device is also considered. A similar operation is performed in Line 3, where the products `weight*value` of neighbours (including the current device) are added. The weighted average is then obtained as `totV / totW` and returned by the function.

This function contains two calls to `nbr`, which in turn perform calls to the `exchange` built-in, both with messages of type `num`. XC ensures that the messages from the different communicating routines are correctly dispatched to neighbours, each used only in the corresponding call to `exchange` in the neighbours, thus not mixing values and weights. ◀

XC ensures that the values produced by an exchange are processed by the *corresponding* exchange in the next round, i.e., the exchange in the same position in the AST and in the same stack frame. Considering both the AST *and* the stack frame ensures that exchange operations are correctly aligned also in case of branches, function calls and recursion. Figure 3 demonstrates alignment. Top-left is a tree representation of the XC program in Example 4, accounting for stack frames and children in the AST. The larger box with multiple compartments denotes the AST of a function, considering only `exchange`, `nfold`, and functions using them. Top-right is a system execution. Dotted arrows connect a round (circle) to the next on the same device, and curly arrows denote messages. Within each round we show a tree corresponding to the one top-left. Note that all rounds execute the same tree. Bottom-left zooms into two rounds of different devices evaluating `average` with fully aligned program executions: *corresponding* expressions at the same tree locations interact and consider each other among neighbouring values. Red dashed arrows connecting exchange expressions that belong to different rounds show this interaction. We will discuss partial alignment in the next section, after introducing conditionals. Alignment is a crucial feature in XC because it *enables functional composition of distributed behaviour*, ensuring that messages are transparently dispatched in the correct way, as exemplified in the following.

▶ **Example 5** (Fire detection)**.** Function `closestFire` returns the distance from the closest likely fire (if any), by relying on the simpler functions `average` and `distanceTo`, based on arguments `temperature` and `smoke` which we can assume to be provided by available sensors.

```
1 def closestFire(temperature, smoke) { // has type scheme: (num, num) → num
2   val trust = nfold(+, 1, 1);
3   val hot = average(trust, temperature) > 60;
4   val cloudy = average(trust, smoke) > 10;
5   distanceTo(hot and cloudy)
6 }
```

In Line 2 the function establishes a trust level for the node, which is proportional to the number of neighbours of that node (thus considering central nodes as more relevant), computed as `nfold(+, 1, 1)`. Line 3 checks whether the average temperature, weighted by trust, is above 60 degrees Celsius. Similarly, Line 4 checks whether the average concentration of smoke, also weighted by trust, is above 10%. Finally, Line 5 computes distances to places where both conditions are met (high temperature and smoke) through function `distanceTo`. Several exchange calls are evaluated by both the `average` and `distanceTo` functions: thanks to alignment, the messages processed by each of them are those generated by the same ones in previous rounds of neighbouring devices. ◀

## 2.5 Conditionals

XC supports `if (cond) {e1} else {e2}` conditional expressions. Crucially, their semantics interplays with the communication semantics of XC. Since only the exchange operations in the same position within the AST and stack frame align, with a conditional, an exchange aligns *only* across the devices that take the same branch. Thus, while evaluating an XC sub-expression, we consider only *aligned neighbours*, that are round neighbours which evaluated the same sub-expression (as AST and stack frame). Non-aligned neighbours are never considered in the evaluation of the sub-expression, e.g., for the construction of the $\underline{\texttt{w}}$ of received messages in an `exchange`, or for determining which values of a nvalue should be folded over by a `nfold`. As a result, a conditional expression *splits* the network into two non-communicating sub-networks, each evaluating a different branch without cross-communication.

▶ **Example 6** (Domain-isolated distance computations)**.** Consider a connected network of service requesters and providers. Suppose these nodes are dynamically split into two domains: those involved in local computations (`local`) and those offloading computations (not `local`) to `gateways`, special service providers which provide cloud access. We may want to compute the distance to `gateways` without considering the devices involved in local computations.

```
def distanceToGateways(local, gateway) { // has type scheme: (bool, bool) → num
  if (local) { Infinity } else { distanceTo(gateway) }
}
```

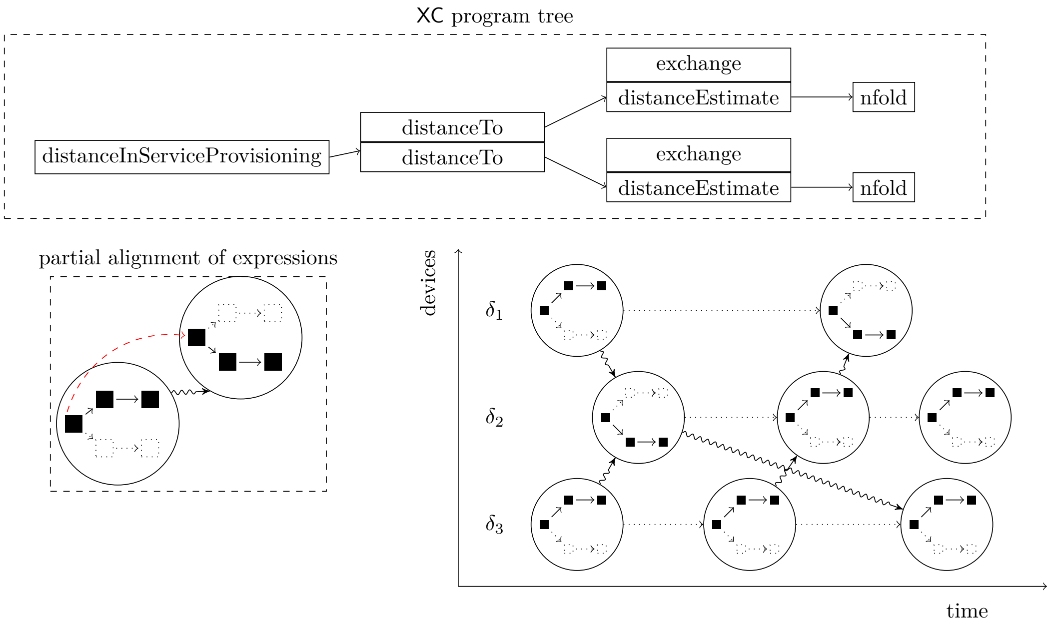



**Figure 4** XC alignment mechanism with conditionals for Example 6.

During a round, the program evaluates to `Infinity` on devices where `local` is true. Such devices are considered "obstacles" to avoid. On devices where `local` is false, the program evaluates `distanceTo(gateway)`, which consist of an exchange-expression (c.f. Example 3). Devices in the `local` group do not compute such exchange expression, and do not contribute to the assessment of distances: `distanceTo` is executed in isolation on non-`local`s.

Now suppose we would like the `local` subgroup to compute distances from local `requester` s, and the other subgroup to still compute distances from `gateway`s, excluding in such computations the devices of the complementary group.

```
// has type scheme: (bool, bool, bool) → bool
def distanceInServiceProvisioning(local, requester, gateway) {
  if (local) { distanceTo(requester) } else { distanceTo(gateway) }
}
```

In this case, in any round, only a single exchange expression is computed, always in the same position in the AST (corresponding to a call of function `distanceTo`). However, the messages exchanged by devices in the `local` group must not be matched with those exchanged by device outside the `local` group, otherwise every device would just compute their distance from the closest local `requester` or non-local `gateway`, which is not the intended behaviour. Luckily, XC grants that this does not happen, as exchange expressions arising from different branches have different stack frames, hence happen in separate interaction domains. ◀

Figure 4 shows partial alignment for Example 6. At the top, we show the program tree for `distanceInServiceProvisioning`. Note that conditionals are not visible here. Bottom-right, we show a system execution: in each round, only one of the `distanceTo` branches is executed – the branch that has *not* been evaluated is dashed. Bottom-left, we zoom into two rounds of devices that align only partially: they evaluate some common expression which is fully aligned (red dashed arrow), then follow a different branch where there is no alignment. Notice that alignment occurs on the execution of function `distanceInServiceProvisioning` but no actual data is exchanged (since no evaluated `exchange` or `nfold` expression is aligned).

### 2.6 Fault tolerance in XC

The abstractions discussed so far allow and encourage developers to write XC programs that are resilient to failures. In case a node fails or a message gets lost in inter-node communication, the `exchange` handles the failure transparently from programmers: the node simply does not show up among the neighbours of a given node in the next alignment. With `exchange`, developers specify the logic to *collectively* operate over the neighbours' messages, and make no assumptions on their number or identity, while being encouraged to express the behaviour homogeneously through point-wise operations and `nfold`. As a result, in XC it is idiomatic to write programs with implicit fault tolerance and resilience to devices that dynamically join and leave the set of neighbours (e.g., because they physically change location), transparently from programmers. Programming resilient behaviour can also take advantage of functional composition: simpler resilient blocks can be composed together, building complex applications while retaining fault-tolerance. However, it is important to note that XC does not provide guarantees on fault tolerance by itself. Being a Turing-complete language, non-resilient behaviour can inevitably be programmed, although mostly non-idiomatically: guarantees on idiomatic subsets of the language may be provided, as briefly discussed in Section 4.

## 3 XC at Work

We now show XC in action by means of example applications in areas like WSNs, IoT, and large-scale CPS. The examples are chosen to *(i)* highlight how composition in XC, enabled by alignment, allows programmers to divide and incrementally deal with the complexity of expressing distributed adaptive behaviour; and *(ii)* show that the expressiveness of XC enables the encoding of advanced algorithms (e.g., with self-organisation properties); and *(iv)* present reusable components used later in our evaluation (Section 6).

▶ **Example 7** (Gossip)**.** The function `gossipEver` spreads the information associated to an event (e.g., pressing a button) through a network. It consists of a single exchange expression executed on every device.

```
1 def gossipEver(event) { // has type scheme: (bool) → bool
2   exchange( False, (n) => retsend nfold(or, n, self(n) or event) )
3 }
```

The first argument of the `exchange` (Line 2) sets the initial value to `False` for $\underline{n}$ (and thus for newly-connected devices, including for the current device in its first round). The second argument is a lambda, whose parameter $\underline{n}$ is the nvalue representing the gossips of neighbouring devices (including the current device itself, for which $\underline{n}$ includes the gossip value in its previous round). Function `nfold` collapses the neighbours' gossips through binary operation `or` (checking whether there is *any* gossip equal to true), with the starting value `self(n) or event` which is true if either the current device had a true gossip in its previous round (i.e., `self(n)` is true) or a true value is fed right now (`event`). The resulting value, the new gossip for the device, is both returned by the function and sent to each neighbour. ◀

Notice that the `gossip` function is agnostic to the network structure and it avoids explicit message management. Its repeated application by a (possibly mobile) network of devices realises the expected behaviour, returning true in every device after a button has been pressed anywhere in the network *as soon as possible*, that is, as soon as the fastest chain of messages from the originating event is able to reach the device.

This function is fully decentralised and every device executes the same logic. Yet, `gossip` only spreads a Boolean event, and once the gossip becomes true, there is no way to flip it to false again. Arbitrary data types and reversibility, require one to break symmetry: some devices (*leaders*) act as sources of truth, and the others will receive their most recent data through a broadcast routine, such as the following.

▶ **Example 8** (Broadcast)**.** Function `broadcast` below implements the propagation of the `value` at nodes of minimal `dist` outwards, along minimal paths ascending `dist`. We assume that `dist` is produced by a function such as `distanceTo` (Example 3).

```
1  def broadcast(dist, value, null) { // has type scheme: (num, A, A) → A
2    val selfRank = (dist, uid);
3    val nbrRank = nbr(selfRank, selfRank);
4    val bestRank = nfold(min, nbrRank, selfRank);
5    val parent = nbrRank == bestRank;
6    exchange( value, (n) =>
7      val selfKey = (value==null, selfRank);
8      val nbrKey = (n==null, nbrRank);
9      val res = snd(nfold( min, (nbrKey, n), (selfKey, value) ));
10     return res
11     send mux(nbr(False, parent), res, null)
12   )
13 }
```

First each device identifies a single *parent device*, as the neighbour having the minimal *rank*, computed in `bestRank` (Line 4). Such rank is a pair of `dist` and `uid` (Line 2), ordered lexicographically, ensuring that the parent is the neighbour of minimal distance to the knowledge source (using `uid` to break ties). The chosen parent is encoded as the only neighbour for which a true value is present in nvalue `parent` (Line 5).

Then, an `exchange` expression sorts out the broadcast received from parent devices, propagating the result to children. The value of the device for the current round is computed in `res` (Line 9), and is taken from the neighbour with the minimum *key*, i.e., minimum rank for a non-`null` value (we assume that `False < True`). For the current device, we use the argument `value` (Line 7). For neighbours, we use the value received from them in `n` (Line 8). The resulting value `res` is returned by exchange and by the whole function, (Line 10). This (possibly band-consuming) value is sent *only* to neighbours which selected the current device as parent, that is, neighbours where `nbr(False, parent)` is true. Every other neighbour receives `null` instead (possibly lighter to transmit): the selection over values and nulls is performed *per-neighbour* by built-in operator `mux` (Line 11). ◀

The function `broadcast` above uses differentiated messages to neighbours to reduce the network load. This result is achieved by sending values only to the neighbours that actually need them, using placeholder `null` values for the others. In case the message propagation does not need to reach every device of the network, but only some targets, this load can be further reduced by restricting the broadcast into a *channel*, as we explain next.

▶ **Example 9** (Broadcast into a Channel)**.** The function `channelBroadcast` selects a region `channel` of a given `width` connecting a `source` device with a destination device `dest`, and performing a broadcast within the region.

```
// has type scheme: (bool, bool, num, A, A) → A
def channelBroadcast(source, dest, width, value, null) {
  val ds = distanceTo(source);
  val dd = distanceTo(dest);
  val channel = ds + dd <= broadcast(ds, dd, Infinity) + width;
  if (channel) { broadcast(ds, value, null) } else { null }
}
```

The channel region is computed through the geometrical definition of ellipse (Line 5): the sum of the distances `ds` towards source and `dd` towards destination (computed by `distanceTo`, Lines 3-4) should surpass the distance between source and destination by at most `width` for devices in the channel. The distance between source and destination is obtained through `broadcast(ds, dd, Infinity)`: the parameter `ds` of the broadcast defines that values should be propagated from the `source` outwards; and the value propagated is the parameter `dd`, as it is evaluated in the source (and thus the distance between source and destination). Then, a conditional is used to selectively broadcast the `value` in the source outwards only in the channel region – `null` elsewhere (Line 6). ◀

The example illustrates functional composition: `channelBroadcast` composes several instances of `distanceTo` (Example 3) and `broadcast` (Example 8) to realise a more complex behaviour. Also, the composition preserves the self-organising properties of its constituent parts, hence it able to automatically adapt to changes in `source`, `dest`, `width`, and topology (because, e.g., of mobility or failure).

So far, we have presented functions to build a communication structure to disseminate information over the network. Yet, we haven't addressed the problem of *collecting* such information, especially in the non-trivial case where it is obtained by inspecting the whole network (or part of it).

▶ **Example 10** (Information collection)**.** The `collect` algorithm (inspired by [8]) aggregates the `value` *currently* present in the network, via an arithmetic or an idempotent accumulator, progressively in a network towards a source node—identified as the zero-value of a gradient `dist` (cf. Example 3). The result is updated when values change, unlike Example 7 where a `true` cannot revert to `false`.

```
1  def weight(dist, radius) { // has type scheme: (num,num)→num
2    max(dist-nbr(0,dist),0)*(radius-senseDist)
3  }
4  def normalise(w) { // has type scheme: (num) → num
5    w / nfold(+, w, 0)
6  }
7  // has type scheme: (num,num,A,(A,A) → A,(A,num) → A) → A
8  def collect(dist, radius, value, accumulate, extract) {
9    exchange( value, (n) =>
10     val loc = accumulate(n, value); // local estimate
11     return loc
12     send extract(loc, normalize(weight(dist, radius)))
13   )
14 }
```

The `exchange` construct (Line 9) handles neighbour-to-neighbour propagation of partial accumulates. First, it applies `accumulate` (Line 10) to aggregate the local `value` with the received partial accumulates `n` into `loc`; this is the result of `collect` (Line 11). In other words, the idea is that the local partial accumulate is obtained by accumulating the partial accumulates of neighbours. Then, it computes a normalised weight (Line 12), via functions `weight` and `normalise`, measuring neighbour reliability, using this weight to `extract` from `loc` the partial accumulates to send to neighbours (Line 12). Function `weight` (Line 1) is parametrised by a gradient value `dist` and value `radius` representing the maximum communication range for neighbour interaction; so, the expression is non-negative and the computed weight is larger for neighbours farther from the communication boundaries (i.e., less likely to be lost as neighbours) and closer to the source of the collection. In `normalise` (Line 4), normalisation of weights `w` is achieved by dividing the computed weights for neighbours by the sum of the neighbours' weights. Depending on the nature of the

```
def smartC(isDetector, isOpsCentre, channelWidth, inspectionRadius, commRadius,
           localWarning, warningThreshold, localLog, nullLog, logCat) {
  val detectorDist = distanceTo(isDetector);
  val inspected = detectorDist < inspectionRadius;
  val nullReport = (uid, 0, nullLog);
  val report = if (inspected) {
    val (sumWarning,numNodes) = collect(detectorDist, commRadius, (localWarning, 1.0),
      (v, l) => (nfold(+, fst(v), fst(l)), nfold(+, snd(v), snd(l))),
      (v, w) => (fst(v)*w, snd(v)*w)
    );
    val meanWarning = sumWarning / numNodes;
    val localWarning = meanWarning > warningThreshold;
    val warning = broadcast(detectorDist, localWarning, False);
    val logs = if (warning) {
        collect(detectorDist, commRadius, localLog, logCat, (v, w) => v)
    } else { nullLog };
    (uid, meanWarning, logs)
  } else { nullReport };
  channelBroadcast(isDetector, isOpsCentre, channelWidth, report, nullReport)
}
```

**Figure 5** Possible XC implementation of a smart city monitoring application

aggregation (*arithmetic* or *idempotent*, e.g., sum or minimum), different `accumulate` and `extract` functions are used: in the former case, the value is multiplied by the weight:

```
def accumulate(v, l) { nfold(+, v, l) } // has type scheme: (A, A) → A
def extract(v, w) { v * w }             // has type scheme: (A, num) → A
```

In the latter case, we choose to either send the value or not (also increasing efficiency as in Example 8) depending on whether the weight exceeds a given threshold:

```
def accumulate(v, l) { nfold(min, v, l) } // has type scheme: (A, A) → A
def extract(v, w) { mux(w >= 0.25, v, Infinity) } // has type scheme: (A, num) → A
```

Improvements over [8] are both stylistic (cleaner code) and in the precision of weights, since in [8] they had to be indirectly (and approximately) deduced on the receiving end. ◀

▶ **Example 11** (Smart City Monitoring). We consider SmartC, a scenario of smart city monitoring, where devices cooperate with neighbours to process and relay information in the distributed system. This is achieved by the collective execution of an XC program. The system consists of *detectors*, non-mobile nodes (e.g., smart traffic lights) that collect in a bounded surrounding area the contributions of other possibly mobile devices that we call *data-providers* (e.g., buses or people with wearables). Data-providers exhibit a local *warning value*, which signals a need for intervention. Detectors collect warning values and compute a *mean warning* in their area: when the mean warning exceeds some threshold, then they also collect logs from data-providers and dispatch collected data towards the closest *operations centre*. The operations centre might be several hops away from the source, so we want to "broadcast" data hop-by-hop along a short "path" of devices—but without flooding the whole network. The system *(i)* collects and routes data from nodes closer than a certain range towards the closest detector; *(ii)* lets detectors compute the mean levels of warning of the corresponding areas; *(iii)* lets detectors collect and aggregate logs if their mean warning exceeds a certain threshold; and *(iv)* creates self-healing broadcast channels from detectors to the closest operations centres. This logic is implemented by function `smartC` (Figure 5), which reuses `distanceTo`, `collect`, `broadcast` and `channelBroadcast` (Examples 3 and 8–10).

**Syntax:**

$$
\begin{array}{lll}
\mathtt{e} ::= \mathtt{x} \mid \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} \mid \mathtt{e}(\overline{\mathtt{e}}) \mid \mathtt{val}\ \mathtt{x} = \mathtt{e}; \mathtt{e} \mid \ell \mid \mathtt{w} & & \text{expression} \\
\mathtt{w} ::= \ell[\overline{\delta} \mapsto \overline{\ell}] & & \text{nvalue} \\
\ell ::= \mathtt{b} \mid \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} \mid \mathtt{c}(\overline{\ell}) & & \text{local literal} \\
\mathtt{b} ::= \mathtt{exchange} \mid \mathtt{nfold} \mid \mathtt{self} \mid \mathtt{updateSelf} \mid \mathtt{uid} \mid \ldots & & \text{built-in function}
\end{array}
$$

**Free variables of an expression:**

$$
\mathsf{FV}(\mathtt{x}) = \{\mathtt{x}\} \quad \mathsf{FV}(\ell) = \mathsf{FV}(\mathtt{w}) = \emptyset \quad \mathsf{FV}(\mathtt{fun}\ \mathtt{x}_0(\mathtt{x}_1, \ldots, \mathtt{x}_n)\{\mathtt{e}\}) = \mathsf{FV}(\mathtt{e}) \setminus \{\mathtt{x}_0, \ldots, \mathtt{x}_n\}
$$

$$
\mathsf{FV}(\mathtt{e}_0(\mathtt{e}_1, \ldots, \mathtt{e}_n)) = \bigcup_{i=0..n} \mathsf{FV}(\mathtt{e}_i) \qquad \mathsf{FV}(\mathtt{val}\ \mathtt{x} = \mathtt{e}; \mathtt{e}') = \mathsf{FV}(\mathtt{e}) \cup \mathsf{FV}(\mathtt{e}') \setminus \{\mathtt{x}\}
$$

**Syntactic sugar:**

$$
\begin{array}{rl}
(\overline{\mathtt{x}})\ \texttt{=>}\ \mathtt{e} & ::= \mathtt{fun}\ \mathtt{y}(\overline{\mathtt{x}})\{\mathtt{e}\} \ \textit{where } \mathtt{y} \textit{ is a fresh variable} \\
\mathtt{def}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} & ::= \mathtt{val}\ \mathtt{x} = \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}; \\
\mathtt{if}(\mathtt{e})\{\mathtt{e}_\top\}\ \mathtt{else}\ \{\mathtt{e}_\bot\} & ::= \mathtt{mux}(\mathtt{e}, ()\ \texttt{=>}\ \mathtt{e}_\top, ()\ \texttt{=>}\ \mathtt{e}_\bot)()
\end{array}
$$

**Figure 6** Syntax (top), free variables (middle) and syntactic sugar (bottom) for FXC expressions

Function `smartC` is defined in terms of local values representing parameters for the algorithm (e.g., `warningThreshold`) or varying inputs (e.g., `localLog`, which denotes a set of log items for a node), which can be thought of as provided by sensors and may change dynamically. The algorithm works as follows. First, a gradient of distances from detectors is computed in the system (Line 3). The nodes that are `inspected` are only those for which the gradient value is less than `inspectionRadius` (Line 4). Then, two different behaviours are defined based on whether a node is inspected or not (Line 6). Nodes not inspected just return `nullReport` (Lines 5 and 18). In the domain of inspected nodes, including the detector, a collection process is activated (Line 7 to 10) in order to let the detector obtain the sum of warning and the number of devices in the area. With such information, the detector can process the mean warning (Line 11) and decide whether the warning level is high (Line 12): such a decision (warning significance) is broadcast from the detector to the rest of the area (Line 13), as a kind of notification to the devices in the surroundings. Also, depending on whether the warning level is high (Line 14 to 16), it either collects the logs from all the nodes in the area (Line 15), or not. In any case, a broadcast on a channel is performed to resiliently communicate the report (set of logs) from the detector to the operations centre (Line 19). ◀

## 4 Formalisation of XC

In this section we present a formalisation of the core concepts introduced in this paper through Featheweight XC (FXC), a minimal calculus for XC. By virtue of its minimality, FXC is particularly convenient for proving properties both of the language as a whole and of algorithms and fragments of it, such as: type soundness and determinism with respect to let-polymorphic typing, denotational characterisation of expressions as space-time values [7], with functional compositionality of global behaviour. We further discuss XC expressivity and resilience properties (inherited from results in literature) in Section 7.

### 4.1 Syntax

Figure 6 (top) shows the syntax of FXC. As in [34], the overbar notation indicates a (possibly empty) sequence of elements, e.g., $\overline{x}$ is short for $x_1, \ldots, x_n$ $(n \geq 0)$. Note that the syntax induces a standard functional language, with no peculiar features for distribution: distribution is nonetheless apparent in the operational semantics. An FXC *expression* $\mathtt{e}$ can be either:

- a *variable* $\mathtt{x}$;
- a (possibly recursive) *function* $\mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$, which may have free variables;
- a *function call* $\mathtt{e}(\overline{\mathtt{e}})$;
- a *let-style* expression $\mathtt{val}\ \mathtt{x} = \mathtt{e}; \mathtt{e}$;
- a *local literal* $\ell$, that is either a built-in function $\mathtt{b}$, a defined function $\mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$ *without* free variables, or a data constructor $\mathtt{c}$ applied to local literals (possibly none);
- an *nvalue* $\mathtt{w}$, as described in Section 2.2.

FXC can be typed using standard let-polymorphism for higher-order languages, without distinguishing between types for local values and types for neighbouring values. This is accomplished by promoting local values to nvalues, and designing constructs and built-in functions of the language to always accept nvalues for their arguments (more details on this in Section 4.2, Device semantics). As local and neighbouring types are not distinguished by FXC, in this section we avoid underlying neighbouring values and their types. Free variables are defined in a standard way (Figure 6, middle), and an expression $\mathtt{e}$ is *closed* if $\mathsf{FV}(\mathtt{e}) = \emptyset$. Programs are closed expressions without nvalues as sub-expressions. Indeed, nvalues only arise in computations, and are the only values produced by evaluating (closed) expressions.

The syntax in Figure 6 (top) diverges partially from the one used in Sections 2 and 3. However, the full syntax of XC can be recovered by defining missing constructs as syntactic sugar. Besides some standard simplifications (infix notation for binary operators, omitted parenthesis in 0-ary constructors, implicit `pair` constructor), some non-trivial encoding is described in Figure 6 (bottom). In particular, lambda expressions can be converted into fun-expressions with a fresh name, and defined functions can be encoded as a let expression binding the function name. Branching can be encoded by abstracting the code in the branches, selecting one of them with the `mux` operator and then applying it.

## 4.2 Operational semantics

The operational semantics is defined as *(i)* a big-step *device semantics*, providing a formal account of the computation of a device within one round; and *(ii)* a small-step *network semantics*, formalising how different device rounds communicate.

**Device semantics.** Figure 7 presents the device semantics, formalised by judgement $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \mathtt{w}; \theta$, to be read as "expression $\mathtt{e}$ evaluates to nvalue $\mathtt{w}$ and value-tree $\theta$ on device $\delta$ with respect to sensor values $\sigma$ and value-tree environment $\Theta$", where:

- $\mathtt{w}$ is called the *result* of $\mathtt{e}$;
- $\theta$ is an ordered tree with nvalues on some nodes (cf. Figure 7 (top)), representing messages to be sent to neighbours by tracking the nvalues produced by exchange-expressions in $\mathtt{e}$, and the stack frames of function calls;
- $\Theta$ collects the (non expired) value-trees received by the most recent firings of neighbours of $\delta$, as a map $\delta_1 \mapsto \theta_1$, ..., $\delta_n \mapsto \theta_n$ $(n \geq 0)$ from device identifiers to value-trees.

We assume every function expression $\mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$ occurring in the program is annotated with a unique name $\tau$ before the evaluation starts. Then, $\tau$ will be the name for the annotated function expression $\mathtt{fun}^{\tau}\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$, and $\mathtt{b}$ the name for a built-in function $\mathtt{b}$.

The syntax of value-trees and value-tree environments is in Fig. 7 (top). The rules for judgement $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \mathtt{v}; \theta$ (Fig. 7, middle) are standard for functional languages, extended to evaluate a sub-expression $\mathtt{e}'$ of $\mathtt{e}$ w.r.t. the value-tree environment $\Theta'$ obtained from $\Theta$ by extracting the corresponding subtree (when present) in the value-trees in the range of

**Auxiliary definitions:**

$$\theta \;::=\; \langle\overline{\theta}\rangle \;\mid\; \mathtt{w}\langle\overline{\theta}\rangle \qquad \text{value-tree} \qquad\qquad \sigma \quad \text{sensor state}$$
$$\Theta \;::=\; \overline{\delta} \mapsto \overline{\theta} \qquad \text{value-tree environment} \qquad\qquad \delta \quad \text{device identifier}$$

$$\pi_i(\langle\theta_1, \dots, \theta_n\rangle) = \theta_i \qquad \pi_i(\mathtt{w}\langle\theta_1, \dots, \theta_n\rangle) = \theta_i \qquad \pi_i(\overline{\delta} \mapsto \overline{\theta}) = \overline{\delta} \mapsto \pi_i(\overline{\theta})$$

$$\overline{\delta} \mapsto \overline{\theta}\,|_{\mathtt{f}} = \left\{\delta_i \mapsto \theta_i \;\mid\; \theta_i = \mathtt{w}\langle\overline{\theta}'\rangle,\; \mathit{name}(\mathtt{w}(\delta_i)) = \mathit{name}(\mathtt{f})\right\} \quad \begin{array}{l}\mathit{name}(\mathtt{b}) = \mathtt{b}\\ \mathit{name}(\mathtt{fun}^{\tau}\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}) = \tau\end{array}$$

**Evaluation rules:** $\boxed{\delta;\sigma;\Theta \vdash \mathtt{e} \Downarrow \mathtt{w};\theta}$

$$[\text{E-NBR}]\;\frac{}{\delta;\sigma;\Theta \vdash \mathtt{w} \Downarrow \mathtt{w};\langle\rangle} \qquad [\text{E-VAL}]\;\frac{\delta;\sigma;\pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \mathtt{w}_1;\theta_1 \qquad \delta;\sigma;\pi_2(\Theta) \vdash \mathtt{e}_2[\mathtt{x} := \mathtt{w}_1] \Downarrow \mathtt{w}_2;\theta_2}{\delta;\sigma;\Theta \vdash \mathtt{val}\ \mathtt{x} = \mathtt{e}_1; \mathtt{e}_2 \Downarrow \mathtt{w}_2;\langle\theta_1,\theta_2\rangle}$$

$$[\text{E-LIT}]\;\frac{}{\delta;\sigma;\Theta \vdash \ell \Downarrow \ell[];\langle\rangle}$$

$$[\text{E-APP}]\;\frac{\delta;\sigma;\pi_{i+1}(\Theta) \vdash \mathtt{e}_i \Downarrow \mathtt{w}_i;\theta_i \quad \text{for all } i \in 0,\dots,n \qquad \delta;\sigma;\pi_{n+2}(\Theta\,|_{\mathtt{f}}) \vdash \mathtt{f}(\mathtt{w}_1,\dots,\mathtt{w}_n) \Downarrow^{*} \mathtt{w}_{n+1};\theta_{n+1} \text{ where } \mathtt{f} = \mathtt{w}_0(\delta)}{\delta;\sigma;\Theta \vdash \mathtt{e}_0(\mathtt{e}_1,\dots,\mathtt{e}_n) \Downarrow \mathtt{w}_{n+1};\mathtt{f}[]\langle\theta_0,\dots,\theta_{n+1}\rangle}$$

**Auxiliary evaluation rules:** $\boxed{\delta;\sigma;\Theta \vdash \mathtt{f}(\overline{\mathtt{w}}) \Downarrow^{*} \mathtt{w};\theta}$

$$[\text{A-FUN}]\;\frac{\delta;\sigma;\Theta \vdash \mathtt{e}[\mathtt{x} := \mathtt{fun}^{\tau}\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}, \overline{\mathtt{x}} := \overline{\mathtt{w}}] \Downarrow \mathtt{w};\theta}{\delta;\sigma;\Theta \vdash \mathtt{fun}^{\tau}\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}(\overline{\mathtt{w}}) \Downarrow^{*} \mathtt{w};\theta} \qquad [\text{A-UID}]\;\frac{}{\delta;\sigma;\Theta \vdash \mathtt{uid}() \Downarrow^{*} \delta;\langle\rangle}$$

$$[\text{A-XC}]\;\frac{\Theta = \overline{\delta} \mapsto \overline{\mathtt{w}}\langle\dots\rangle \quad \mathtt{w} = \mathtt{w}_i[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)] \qquad \delta;\sigma;\pi_1(\Theta) \vdash \mathtt{w}_f(\mathtt{w}) \Downarrow (\mathtt{w}_r,\mathtt{w}_s);\theta}{\delta;\sigma;\Theta \vdash \mathtt{exchange}(\mathtt{w}_i,\mathtt{w}_f) \Downarrow^{*} \mathtt{w}_r;\mathtt{w}_s\langle\theta\rangle} \qquad [\text{A-SELF}]\;\frac{}{\delta;\sigma;\Theta \vdash \mathtt{self}(\mathtt{w}) \Downarrow^{*} \mathtt{w}(\delta);\langle\rangle}$$

$$[\text{A-FOLD}]\;\frac{\Theta = \delta_1 \mapsto \theta_1,\dots,\delta_n \mapsto \theta_n \quad \ell_0 = \mathtt{w}_3(\delta) \qquad \delta;\sigma;\emptyset \vdash \mathtt{w}_1(\ell_{i-1},\mathtt{w}_2(\delta_i)) \Downarrow \ell_i[];\theta \text{ if } \delta_i \neq \delta \text{ else } \ell_i = \ell_{i-1} \quad \cdots}{\delta;\sigma;\Theta \vdash \mathtt{nfold}(\mathtt{w}_1,\mathtt{w}_2,\mathtt{w}_3) \Downarrow^{*} \ell_n[];\langle\rangle}$$

**Figure 7** Device (big-step) operational semantics of FXC

$\Theta$. This *alignment* process is modelled by the auxiliary "projection" functions $\pi_i$ (for any positive natural number $i$) (Fig. 7, top). When applied to a value-tree $\theta$, $\pi_i$ returns the $i$-th sub-tree $\theta_i$ of $\theta$. When applied to a value-tree environment $\Theta$, $\pi_i$ acts pointwise on the value-trees in $\Theta$.

The alignment process ensures that the value-trees in the environment $\Theta$ always correspond to the evaluation of the same sub-expression currently being evaluated. To ensure this match holds (as said before, of the stack frame and position in the AST), in the evaluation of a function application $\mathtt{f}(\overline{\mathtt{w}})$, the environment $\Theta$ is reduced to the smaller set $\Theta\,|_{\mathtt{f}}$ of trees which corresponded to the evaluation of a function with the same *name* (as defined in Fig. 7 (top)).

Rule [E-NBR] evaluates an nvalue $\mathtt{w}$ to $\mathtt{w}$ itself and the empty value-tree. Rule [E-LIT] evaluates a local literal $\ell$ to the nvalue $\ell[]$ and the empty value-tree. Rule [E-VAL] evaluates a val-expression, by evaluating the first sub-expression with respect to the first sub-tree of the environment obtaining a result $\mathtt{w}_1$, and then the second sub-expression with respect to the second sub-tree of the environment, after substituting the variable $\mathtt{x}$ with $\mathtt{w}_1$.

Rule [E-APP] is standard eager function application: the function expression $\mathtt{e}_0$ and each argument $\mathtt{e}_i$ are evaluated w.r.t. $\pi_{i+1}(\Theta)$ producing result $\mathtt{v}_i$ and value-tree $\theta_i$. Then, the function application itself is demanded to the auxiliary evaluation rules, w.r.t. the last sub-tree of the trees corresponding to the same function: $\pi_{n+2}(\Theta\,|_{\mathtt{f}})$. The auxiliary rule [A-FUN] handles the application of fun-expression, which evaluates the body after replacing the

**Network configuration (sensors/environment/result fields) and action labels:**

$$\begin{array}{llll}
\Sigma ::= \overline{\delta} \mapsto \overline{\sigma} & \text{sensors field} & N ::= \langle \Sigma; \Psi; \psi \rangle & \text{network configuration} \\
\Psi ::= \overline{\delta} \mapsto \overline{\Theta} & \text{environment field} & & \\
\psi ::= \overline{\delta} \mapsto \overline{\mathtt{w}} & \text{result field} & \mathit{act} ::= \delta \;\big|\; \delta\delta' \;\big|\; \mathit{conf} & \text{action label}
\end{array}$$

**Notations for restriction and update of a sensors/environment/result field $m$:**

$$m|_X = m' \quad \text{s.t.} \quad \mathbf{dom}(m') = \mathbf{dom}(m) \cap X \text{ and } m'(\delta) = m(\delta)$$

$$m[m'] = m'' \text{ s.t. } \mathbf{dom}(m'') = \mathbf{dom}(m) \cup \mathbf{dom}(m'), \;\; m''(\delta) = \begin{cases} m'(\delta) \text{if } \delta \in \mathbf{dom}(m') \\ m(\delta) \text{ otherwise} \end{cases}$$

**Transition rules:** $\boxed{N \xrightarrow{act} N}$

$$\text{[N-FIRE]}\; \frac{\Theta = \text{filter}(\Psi(\delta)) \quad \delta; \Sigma(\delta); \Theta \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \mathtt{w}; \theta \quad \Theta' = \Theta[\delta \mapsto \theta]}{\langle \Sigma; \Psi; \psi \rangle \xrightarrow{\delta} \langle \Sigma; \Psi[\delta \mapsto \Theta']; \psi[\delta \mapsto \mathtt{w}] \rangle}$$

$$\text{[N-RECV]}\; \frac{\theta = \Psi(\delta)(\delta) \quad \Theta' = \Psi(\delta')[\delta \mapsto \theta]}{\langle \Sigma; \Psi; \psi \rangle \xrightarrow{\delta\delta'} \langle \Sigma; \Psi[\delta' \mapsto \Theta']; \psi \rangle} \qquad \text{[N-CONF]}\; \frac{\overline{\delta} = \mathbf{dom}(\Sigma') \quad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle \Sigma; \Psi; \psi \rangle \xrightarrow{conf} \langle \Sigma'; \Psi_0[\Psi|_{\overline{\delta}}]; \psi|_{\overline{\delta}} \rangle}$$

**Figure 8** Network (small-step) operational semantics of FXC

arguments $\overline{\mathtt{x}}$ with their provided values $\overline{\mathtt{w}}$, and the function name $\mathtt{x}$ with the fun-expression itself. Rules [A-UID] and [A-SELF] trivially encode the behaviour of the `uid` and `self` built-ins. Rule [A-XC] evaluates an exchange-expression, realising the behaviour described at the beginning of Section 2.3. Notation $\mathtt{w}_1[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)]$ is used to represent the nvalue $\mathtt{w}_1$ after the update for each $i$ of the message for $\delta_i$ with the custom message $\mathtt{w}_i(\delta)$. The result is fed as argument to function $\mathtt{w}_f$: the first element of the resulting pair is the overall result, while the second is used to tag the root of the value-tree. Rule [A-FOLD] encodes the `nfold` operators. First, the domain of $\Theta$ is inspected, giving a (sorted) list $\delta_1, \dots, \delta_n$. An initial local value $\ell_0$ is set to the "self-message" of the third argument. Then, a sequence of $\ell_i$ is defined, each by applying function $\mathtt{w}_1$ to the previous element in the sequence and the value $\mathtt{w}_2(\delta_i)$ (skipping $\delta$ itself). The final result $\ell_n$ is the result of the application, with empty value-tree. Auxiliary rules for the other available built-in functions are standard, do not depend on the environment, hence have been omitted.

**Network semantics.** The evolution of a whole network of devices executing a program $\mathtt{e}_{\mathtt{main}}$ is formalised by transitions $N \xrightarrow{act} N'$, which reads *"network configuration $N$ evolves to network configuration $N'$ by a transition with label act"*. The syntax of network configurations and action labels is in Figure 8 (top). A network configuration $N$ is a triple $\langle \Sigma; \Psi; \psi \rangle$, where:

- $\Sigma$ maps each device $\delta$ of the network to a sensors status $\sigma$, representing the status of sensors of $\delta$ at a given time (for any choice of a representation of sensor status $\sigma$);
- $\Psi$ maps each device $\delta$ of the network to a value-tree environment $\Theta$, collecting the (non expired) value-trees received by the most recent firings of neighbours of $\delta$;
- $\psi$ is a partial mapping that, at any given time, maps devices $\delta$ of the network to the nvalue $\mathtt{w}$ produced by their most recent firings (if any such firing already happened).

*We remark that, for each device $\delta$, the sensors status $\Sigma(\delta)$, the value-tree environment $\Psi(\delta)$ and the nvalue $\psi(\delta)$ are locally stored in the device $\delta$ – there is no global memory.*

Each transition $N \xrightarrow{act} N'$ consists of one of these three different evolution steps:

- if $\mathit{act} = \delta$, it formalises the round of device $\delta$, and the memorisation of the resulting nvalue $\mathtt{w}$ and value-tree $\theta$ in the device's local store;

- if $act = \delta\delta'$ with $\delta \neq \delta'$, it formalises that device $\delta'$ receives a value-tree $\theta$ from $\delta$;
- if $act = conf$, it formalises an overall change of the network configuration as (possible) change of sensor status of devices and (possible) entering/leaving of devices in the network.

A sequence of transitions $\langle\emptyset;\emptyset;\emptyset\rangle \xrightarrow{act_1} \dots \xrightarrow{act_n} N_n$ thus represents the operational evolution of a network. The transition rules of the semantics of a program $\mathtt{e_{main}}$ are given in Figure 8 (bottom). Rule [N-FIRE] formalises a computation round of device $\delta$: given the locally-available sensors status $\Sigma(\delta)$ and value-tree environment filtered out of expired value-trees $\Theta = \mathrm{filter}(\Psi(\delta))$, it uses the device semantics judgement to obtain the nvalue $\mathtt{w}$ and value-tree $\theta$ produced by the round. Then, it uses $\mathtt{w}$ to update $\psi(\delta)$, and uses $\theta$ to update $\Theta(\delta)$ (thus modelling immediate reception of the self-message). The filtering function $\mathrm{filter}(\cdot)$ is a parameter of the calculus, meant to clear out old stored values from the value-tree environments in $\Psi$, usually based on space/time tags attached to value-trees.

Rule [N-RECV] formalises the reception of a value-tree from device $\delta$ by another device $\delta'$. The message conceptually dispatched is the value-tree $\theta$ corresponding to $\delta$ obtained from the value-tree environment $\Psi(\delta)$ of $\delta$ itself. On the recipient side, the received message $\theta$ is locally associated to $\delta$ in the value-tree environment $\Psi(\delta')$ of $\delta'$. Even though rule [N-RECV] dispatches the same message $\theta$ to any recipient $\delta'$, an optimised implementation could compress received messages by collapsing each received nvalue $\mathtt{w}$ within $\theta$ to the message $\mathtt{w}(\delta')$ for $\delta'$, discarding the rest before storing it in the local memory.

Rule [N-CONF] formalises an update of the sensor status of devices and entering/leaving of devices (auxiliary notations $m\,|_X$ and $m[m']$ are in Fig. 8, second frame, representing domain restriction and pointwise update of maps). Given a new sensors mapping $\Sigma'$, the resulting network configuration contains exactly the devices $\overline{\delta}$ in the domain $\mathbf{dom}(\Sigma')$ of $\Sigma'$. This is achieved by reducing the result field $\psi$ to the new set of devices through $\psi\,|_{\overline{\delta}}$, constructing an environment field $\Psi_0$ mapping every $\overline{\delta}$ to the empty environment $\emptyset$, then reducing the existing environment field $\Psi$ to the new set of devices through $\Psi\,|_{\overline{\delta}}$, and finally using this to overwrite the values in $\Psi_0$. Note that the reboot of a device $\delta$ can be modelled by two applications of rule [N-CONF]: one removing $\delta$ from the network configuration and another re-inserting it. When a device $\delta$ is removed from the network, the content of its local memory (sensors, messages, result) are lost.

## 5 Implementation

We implemented a Scala and a C++ version of XC. The Scala version has been developed as an extension of ScaFi [22], and aims at showcasing the DSL and maximize portability to different platforms, including simulators. The C++ version has been developed as an extension of FCPP [5], and has consequently been integrated into the main FCPP distribution. This version targets performance and devices with limited resources. Running experiments on real IoT devices with the C++ version is still work in progress.

### 5.1 Scala DSL

We provide an implementation of XC as a DSL embedded into the Scala language[3] because of its cross-platform support [30], popularity for building distributed systems [33], and advanced support for internal DSLs [4]. This implementation has been developed as an extension of

[3] The Scala DSL is publicly available under the Apache 2.0 license at `https://github.com/scafi/artifact-2021-ecoop-xc` and permanently as an archived artifact on Zenodo [21].

ScaFi [22]. The DSL is organized into a few core XC constructs and a library of reusable functions. The core constructs (cf. Figure 6) are declared by a Scala trait with the following interface:

```
trait XCLang {
  def branch[T](cond: NValue[Boolean])(th: => NValue[T])(el: => NValue[T]): NValue[T]
  def exchange[T](init: NValue[T])(f: NValue[T] => (NValue[T],NValue[T])): NValue[T]
}
```

The `if`/`else` of XC is modelled as a `branch` function to avoid conflicts with Scala's `if`. The two branches are call-by-name parameters, as usual. A neighbour value is implemented as a class with a default message and a concrete map of messages for other devices.

```
class NValue[T](val defaultMessage: T, val customMessages: Map[ID,T] = Map.empty) {
  def fold[V>:T](init: V)(f: (V,V)=>V): NValue[V] = // ...
  def map2[R,S](other: NValue[R])(m: (T,R)=>S): NValue[S] = // ...
  // more built-ins ... (cf. Figure 2)
}
```

We leverage Scala implicit conversions and extension methods [25], imported by mixing in `XCLib`, to automatically convert values of type `T` to `NValue`s of `T`s and, e.g., to extend `NValue`s of `Numeric`s to accept operators like `+` (to combine nvalues point-wise). An abstract class `XCProgram[T]` requires programmers to override the method `main:T`. Moreover, it exposes methods `sense` and `senseNeighbour` to subclasses for retrieving local and neighbouring values from the execution environment. For instance, the gradient program (Example 3) can be encoded as follows.

```
object gradient extends XCProgram[Double] with XCLib {
  def main =
    exchange(Double.PositiveInfinity)(n =>
      mux(sense[Boolean]("source")){ 0.0 }{
        (n + senseNeighbour("distance")).fold(Double.PositiveInfinity)(Math.min)
      })
}
```

An `XCProgram[T]` models a single local computation. As discussed (Section 2.1), a XC system involves multiple devices repeatedly acquiring context, computing the round, and propagating messages to neighbours. The execution environment provides a context with values from the sensors for the built-in sensing functions (cf. Figure 2) and with the messages from the neighbours. For example, the following code shows the execution on a device:

```
1 while(true) {
2   val sensorData = getData() // implementation-specific
3   val messagesFromNeighbours = getMessages() // implementation-specific
4   val context = Context(sensorData, messagesFromNeighbours)
5   val (output, messageCollection) = gradient.fire(context)
6   process(output) // implementation-specific
7   propagate(messageCollection) // implementation-specific
8 }
```

Note that in this implementation message communication occurs only *before* (Line 3) and *after* (Line 7) the firing (Line 5) to ensure that the `exchange` within the round are all executed atomically w.r.t. the messages that are received and sent by the device (Section 2.1). The details of a system implementation depends on the target deployment. Example deployments that could be implemented include a peer-to-peer network of IoT devices (where each node handles computation and communication with neighbours), a collection of thin IoT devices connected to the cloud (where only sensor and actuator data flows between the IoT nodes and the cloud, which is responsible for running computations and internally handling the

```
FUN bool gossipEver(ARGS, bool event){ CODE
  return nbr(CALL, false, [&](field<bool> n){
    return any_hood(CALL, n) or event;
  });
}
```

**Figure 9** Implementation of the `gossipEver` function in C++/XC.

message passing), or a simulator (where physical and/or logical devices are virtualised). What these implementations must do in order to support a XC system is providing the implementation-specific functions of the listing above: `getData()` to obtain values from the local environment, `getMessages()` to retrieve messages from neighbours (e.g., a peer node may keep them in a buffer, a cloud platform may use an in-memory database service, a simulator may use an ad-hoc map-like data structure), `process()` to drive actuations (e.g., locally on a node, or through a command on a cloud back-end), and `propagate()` to send exported data to neighbours (e.g., through a direct message to the neighbour, or through a write on shared state in simulations or cloud).

## 5.2 C++ DSL

We implemented XC as a C++ DSL[4], by extending FCPP [5]. This implementation is designed for *(i)* efficiency, and *(ii)* custom architectures. For *(i)*, we rely on C++'s compile-time optimization and execution on the bare metal. We also performed careful profiling to manually optimize crucial parts of the library. For neighbouring values, we use `vector<T>` (having two sorted lists for ids and values) from C++ STL—which is more efficient than hash maps for linear folding and point-wise operations. For communication, we serialise messages and pass them to the network driver (for low level devices this is usually a non-standard API where one can configure the byte content of the message and the transmission power). For *(ii)* we exploit that C/C++ compilers are usually available for custom architectures, while also aiming to minimise the amount of dependencies, to ease the deployment. For instance, the implementation includes its own serialisation header, compile-time type inspection utilities, multi-type valued maps, option types, quaternions, tagged tuples, etc.

Compared to the Scala implementation, the embedding of XC into C++ is more verbose, thus requiring additional effort for development (see Figure 9 for a code sample). We are currently working on testing this implementation on several different back-ends, including:

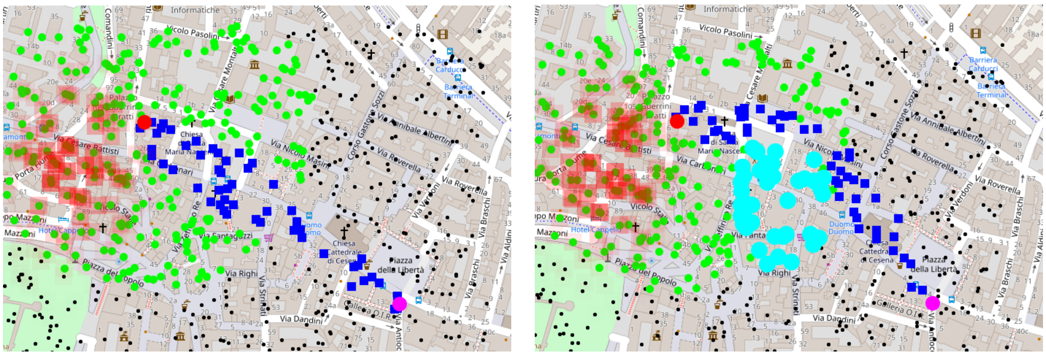
- processing of XC algorithms on large graph-based data in HPC;
- deployment on microcontroller architectures with either Contiki OS or MIOSIX.

No external dependencies are needed for those back-ends.

# 6 Evaluation

In this section, we evaluate XC.[5] The goal is to show that *(RQ1)* the decentralised execution of the XC program on each device results in the desired *collective* behaviour and that *(RQ2)* the overall behaviour can be expressed by composing functions of collective behaviour that

[4] The C++ DSL is publicly available under the Apache 2.0 license at: `https://fcpp.github.io`.
[5] The simulation framework, its description, and instructions for reproducing the experiments are publicly available at `https://github.com/scafi/artifact-2021-ecoop-smartc` and permanently as an archived artifact on Zenodo [20].



**(a)** Communication structures are in place (inspection area and channel from detector to operations centre). Sensors detect some dangerous situation.

**(b)** A blackout destroys the original channel. The channel self-repairs by circumventing the obstacle.

**Figure 10** Two snapshots of the SmartC case study.

correctly combine thanks to alignment. The evaluation does not focus on the efficiency of fault recovering because this aspect is application-dependent – not language-dependent. For instance, the recovery time for a channel depends on the algorithms used to compute distances and broadcasts, relative to the network assumptions.

**SmartC case study.** We consider a simulation of the SmartC scenario described in Example 11, and we implement it both in the Scala and C++ DSLs (the results in this section refer to the Scala implementation). We believe that other application domains, such as cyber-physical systems (CPS) and wireless sensor networks (WSN), would not pose fundamentally different challenges compared to the considered scenario: WSN focus on information flows, which is part of the case study, and CPS emphasize on actuation, which could be a simple variant of the scenario, e.g., where agents move according to the gathered reports. In the simulation setup, 600 devices each running the XC program communicate with every neighbour currently in a 50-metre range once per second. We consider a single detector and a single operations centre. The simulator enables the collection of data exported at the individual nodes (i.e., the program Example 11 is extended with simulation-specific code). We measure, every second, the actual (instantaneous) mean warning in the inspection area (using an oracle, namely a process that can inspect the simulated system at any instant) and the mean warning measured by the operations centre. We consider the average result over 30 simulations varying the actual displacement of devices and scheduling offsets. We inject a blackout event that disconnects a set of devices from the system, hence disrupting the channel. Figure 10 shows two snapshots of the simulation with devices (black dots), detector (red dot), sensors within the area inspected by the detector (green dots), operations centre (magenta dot), and inoperable devices (cyan dots). Semi-transparent red squares denote the warning level locally perceived by sensors. Blue squares are nodes in the channel from detector to operations centre.

**Results.** Figure 11 shows that the mean warning received by the operations centre (blue) during a run approximates the actual warning in the inspected area (magenta). The jags in the second blue wave are due to perturbations (exacerbated by the obstacle) that temporarily destroy the channel in a few simulations, while delays depend on the firing frequency and communication hops (from inspection area edges to detector to operations centre).

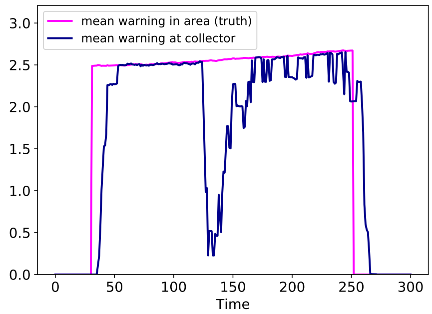



**Figure 11** Execution of SmartC.

For *(RQ1)*, this result shows that the XC program enables the system to self-organise in such a way that the operations centre can acquire the mean warning level aggregated by the detector, in spite of environmental changes and perturbations induced by mobility and failure. For *(RQ2)*, we remark that the self-organising behaviour resulting from the XC program in SmartC is achieved by direct composition of several reusable blocks of collective behaviour, namely `distanceTo`, `broadcast`, `collect`, and `channelBroadcast` (cf. Example 11).

**Comparison with other programming models.** Additionally, to get a sense of the benefit of the XC implementation w.r.t. other programming models, we re-implemented functions `distanceTo` and `channel` (a version of `channelBroadcast` without the final broadcast) with actors and pub-sub[6]. Essentially, the intuition of the XC advantage in terms of expressiveness lies in the implicit declaration of data exchange for each building block usage, instead of the more explicit and verbose message handling/sending (for actors) and topic forging with event consuming/producing (for pub-sub). Despite field calculi have been studied in a series of papers [49], no systematic comparison (e.g. via formal translation) with other approaches has been previously carried out. There are two challenges: *(i)* very few works target the kind of distributed systems (e.g. self-organisation) targeted by XC, hence a comparison needs to consider general-purpose languages and a wide spectrum of software designs; *(ii)* system behaviour unfolds by the interplay of device semantics *and* network semantics (cf. Section 4.2), which are brittle to neatly separate in other approaches. Though, we can here focus on a comparison among *(i)* a pub-sub "idiomatic" solution, $S_{\mathrm{PS}}$, *(ii)* a pub-sub XC-like solution $S_{\mathrm{PSXC}}$ with a design inspired by XC, and *(iii)* an XC solution $S_{\mathrm{XC}}$. This allows us to draw some interesting indications on the compactness that programming in XC can provide.

The core programs are 82, 28, 22 LoC long. In $S_{\mathrm{PS}}$, the logic spreads over multiple subscription handlers, while in $S_{\mathrm{PSXC}}$ and $S_{\mathrm{XC}}$ the core logic is neatly separated. The $S_{\mathrm{PS}}$ version uses 4 handlers (and crucially, any additional field would need a further handler), 2 `send`s, and 4 `publish`es, while $S_{\mathrm{PSXC}}$ uses 2, 1, and 3 resp. Also, $S_{\mathrm{PS}}$ keeps 6 state variables for the input context of a device–$S_{\mathrm{PSXC}}$ only 3. W.r.t. $S_{\mathrm{XC}}$, $S_{\mathrm{PSXC}}$ has a coding overhead due to the topics management and to the more brittle handling of neighbour data, of about 27% more LoC, 73% more words, and 35% more method calls. Finally, the main limitation of $S_{\mathrm{PS}}$ is the loss of compositionality, the inter-dependence between the different computations of fields, and the fragility that stems from the management of change propagation.

[6] The paradigm comparison is publicly available at: `https://github.com/metaphori/aggregate-paradigm-comparison`.

## 7 Related work

We organize related work by first providing a high-level perspective on field-based coordination. Next we describe approaches based on ensembles and attribute-based communication, which are close to our solution but adopt fundamentally different design choices. Finally, we compare in detail with field calculi and briefly discuss abstraction and compositionality.

**Field-based coordination.** Field-based coordination, as a paradigm to develop self-organising systems, originate from two main research areas: *spatial computing* [29], where the idea of *aggregate computing* [16] emerged, and *coordination models and languages* [39]. Two surveys cover these two perspectives. The work in [15] reviews various DSLs ranging from multi-agent modelling to WSNs with respect to how they measure and manipulate space-time, model physical evolution and computation, and (meta-)manipulate computation itself. More recently, [49] outlines the historical development from tuple-based and field-based coordination to field calculi, covering the state of the art and future challenges within aggregate computing research. The latter work also reviews various formalisations of field computations. As discussed later, XC subsumes the constructs of field calculi as of [48, 6] and so has a potential as foundation for field-based coordination, and as *lingua franca* to describe distributed algorithms for large-scale systems, and specifically for self-organisation.

**Ensembles and attribute-based communication.** Recently, field-based coordination is also framed as a paradigm for collective adaptive systems (CAS) [32], which is a further application target for self-organisation techniques in general. There, related approaches include *ensemble-based engineering* [19, 27] and *attribute-based communication* [1]. Ensemble approaches leverage the notion of *ensemble*, i.e., a dynamic group of components typically specified through a membership relationship, for CAS programming. De Nicola et al. propose SCEL [27], a process-algebraic approach where systems are made of components, i.e., processes with an attribute-based interface for addressing their state (knowledge) and evolving by executing actions on predicated groups of target components; actions provide ways to read, retrieve, put information, and to create new components. AbC [1] captures the essence of attribute-based interaction of SCEL: components are (parallel compositions of) processes associated with an attribute environment, and actions are guarded through predicates over such attributes. Attribute-based communication approaches exploit attributes labelling devices and matching mechanisms to dynamically define sets of recipients for multi-casts, to promote coordination in CASs. This is also possible in field calculi, but it is made much simpler by the selective communication mechanism in XC, a key contribution of this paper.

**Field calculi.** Field calculi, surveyed in [49], assume a neighbouring relationship for connectivity and, upon that, enable defining dynamic groups of devices by exploiting branching and recursion. However, interaction is not based on attribute matching but on execution of the same functions (alignment) involving communication constructs like `exchange`.

In the following we compare XC with the *field calculus (FC)* [48, 6], which is the reference model for computational fields [49], also implemented by DSLs like ScaFi [22, 23] and FCPP [5, 13]. FC features two separate kinds of values (and types): *local values* (of *local* type) and *neighbouring values* (of *field* type). XC combines these into a single class of nvalues $\mathtt{v} = \ell[\overline{\delta} \mapsto \overline{\ell}]$. In particular, *local values* are equivalent to nvalues $\ell[]$ without custom messages, and *neighbouring values* are equivalent to nvalues with any valid default message. This unification allows a simpler type system and, crucially, differentiated messages to neighbours.

By interpreting FC values as nvalues, all FC message-exchanging constructs (`nbr`, `rep` [48] and `share` [6]) can be modelled within XC: `nbr` is the same defined function introduced in Section 2.2, just restricted to operate on local values only; `share` corresponds to an `exchange` with `retsend` restricted to operate on local values only; and `rep`($e_1$){(`x`) => $e_2$} can be translated to `exchange`($e_1$, (`x`) => `retsend` $e_2$[`x` := `self`(`x`)]). Notice that the converse translation is not possible, as `nbr`, `rep` or `share` expressions with arguments of neighbouring type have no defined behaviour in FC. Thus, `nbr` and `exchange` in XC are strictly more expressive than their corresponding FC counterparts `nbr` and `share`: they can be used with expressions producing nvalues with custom messages to model differentiated messages.

The properties for subsets of the *field calculus* (FC), as surveyed in [49], include eventual recovery and stabilisation after transient changes (self-stabilisation) [48], independence of the results from the density of devices [17], real-time error guarantees [12], efficient monitorability of spatio-temporal logic properties [9, 11], and ability to express all physically consistent computations (space-time universality) [7]. The fact that every FC program can be encoded within XC, automatically imports all these results into XC and paves the way towards future extensions to XC programs not expressible in FC.

**Abstraction and compositionality.** XC's mechanism to send and receive messages to/from neighbours provides a high-level programming model for message passing which abstracts over failures (cf. Section 2.6) and is reminiscent of shared memory models. Namely: *(i)* nodes work on a fixed snapshot of incoming messages once the round starts (because message exchange occurs only between rounds) and *(ii)* messages can be overwritten or read multiple times until they expire, resulting in a model similar to shared memory. This combination, thanks to the alignment property (a distinctive feature of XC and field calculi, which enables functional composition as illustrated in Sections 2.4 and 2.5), achieves an abstraction level that it is not available in the competing spatial computing approaches (surveyed, e.g., in [15, 49]) or shared memory models (surveyed, e.g., in [42, 43]).

## 8 Conclusion and Outlook

In this paper, we introduce the design of XC, a programming language for homogeneous distributed systems that abstracts over a number of traditional issues in developing distributed applications, including faults, lost messages, and asynchronicity. XC's minimal design features only one communication primitive. We show that despite its simplicity, XC can capture a number of communication patterns in homogeneous distributed systems and it is effective for writing large scale distributed software.

The design of XC, through nvalues and the new semantic construct `exchange`, opens interesting directions for future work. First, we plan to characterise XC programs enjoying two fundamental properties: self-stabilisation [48], and density independence [17], as the ability of a field computation to converge with the density of devices filling space. Second, works such as [48] define combinators, namely, general field functions implementing key behavioural elements of information diffusion, collection, and degradation, the composition of which turns out to define a number of interesting higher-level functions. We plan to devise new such building blocks with XC, e.g. to realise *sparse choice* of leaders [41] and consensus [14]. Finally, we are currently assessing the impact of XC constructs on real-world application programming, thanks to our porting in Scala and C++.

## References


1 Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. Programming interactions in collective adaptive systems by relying on attribute-based communication. *Science of Computer Programming*, 192, 2020. doi:10.1016/j.scico.2020.102428.

2 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Comput.*, 9(1):37–49, 1995. doi:10.1007/BF01784241.

3 Joe Armstrong. Erlang. *Commun. ACM*, 53(9), September 2010. doi:10.1145/1810891.1810910.

4 Cyrille Artho, Klaus Havelund, Rahul Kumar, and Yoriyuki Yamagata. Domain-specific languages with Scala. In *ICFEM*, volume 9407 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2015. doi:10.1007/978-3-319-25423-4_1.

5 Giorgio Audrito. FCPP: an efficient and extensible field calculus framework. In *Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS*, pages 153–159. IEEE Computer Society, 2020. doi:10.1109/ACSOS49614.2020.00037.

6 Giorgio Audrito, Jacob Beal, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Field-based coordination with the share operator. *Logical Methods in Computer Science*, 16(4), 2020. doi:10.23638/LMCS-16(4:1)2020.

7 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. doi:10.1007/978-3-319-92408-3_1.

8 Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Danilo Pianini, and Mirko Viroli. Optimal resilient distributed data collection in mobile edge environments. *Computers & Electrical Engineering*, 2021. doi:10.1016/j.compeleceng.2021.107580.

9 Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Volker Stolz, and Mirko Viroli. Adaptive distributed monitors of spatial properties for cyber-physical systems. *J. Syst. Softw.*, 175:110908, 2021. doi:10.1016/j.jss.2021.110908.

10 Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, and Mirko Viroli. Compositional blocks for optimal self-healing gradients. In *Self-Adaptive and Self-Organizing Systems (SASO), 2017*, pages 91–100. IEEE, IEEE Computer Society, 2017. doi:10.1109/SASO.2017.18.

11 Giorgio Audrito, Ferruccio Damiani, Volker Stolz, Gianluca Torta, and Mirko Viroli. Distributed runtime verification by past-CTL and the field calculus. *J. Syst. Softw.*, 187:111251, 2022. doi:10.1016/j.jss.2022.111251.

12 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. doi:10.1109/RTSS.2018.00013.

13 Giorgio Audrito, Luigi Rapetta, and Gianluca Torta. Extensible 3D simulation of aggregated systems with FCPP. In *24th International Conference on Coordination Models and Languages, Proceedings*, Lecture Notes in Computer Science. Springer, 2022. To appear.

14 Jacob Beal. Trading accuracy for speed in approximate consensus. *Knowledge Eng. Review*, 31(4):325–342, 2016. doi:10.1017/S0269888916000175.

15 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. doi:10.4018/978-1-4666-2092-6.ch016.

16 Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the Internet of Things. *IEEE Computer*, 48(9), 2015. doi:10.1109/MC.2015.261.

17 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*, 12(3):12:1–12:29, 2017. doi:10.1145/3105758.

18 Arne Brutschy, Giovanni Pini, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. Self-organized task allocation to sequentially interdependent tasks in swarm robotics. *Auton. Agents Multi Agent Syst.*, 28(1):101–125, 2014. `doi:10.1007/s10458-012-9212-y`.

19 Tomás Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: an ensemble-based component system. In *Symposium on Component Based Software Engineering (CBSE)*, pages 81–90. ACM, 2013. `doi:10.1145/2465449.2465462`.

20 Roberto Casadei. scafi/artifact-2021-ecoop-smartc: v1.2, 2022. URL: `https://zenodo.org/record/6538822`, `doi:10.5281/ZENODO.6538822`.

21 Roberto Casadei. scafi/artifact-2021-ecoop-xc: v1.2, 2022. URL: `https://zenodo.org/record/6538810`, `doi:10.5281/ZENODO.6538810`.

22 Roberto Casadei, Mirko Viroli, Giorgio Audrito, and Ferruccio Damiani. FScaFi : A core calculus for collective adaptive systems programming. In *ISoLA (2)*, volume 12477 of *Lecture Notes in Computer Science*, pages 344–360. Springer, 2020. `doi:10.1007/978-3-030-61470-6_21`.

23 Roberto Casadei, Mirko Viroli, Giorgio Audrito, Danilo Pianini, and Ferruccio Damiani. Engineering collective intelligence at the edge with aggregate processes. *Eng. Appl. Artif. Intell.*, 97:104081, 2021. `doi:10.1016/j.engappai.2020.104081`.

24 Shane S. Clark, Jacob Beal, and Partha P. Pal. Distributed recovery for enterprise services. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*, pages 111–120. IEEE Computer Society, 2015. `doi:10.1109/SASO.2015.19`.

25 Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 341–360. ACM, 2010. `doi:10.1145/1869459.1869489`.

26 Soura Dasgupta and Jacob Beal. A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In *Decision and Control (CDC), 2016 IEEE 55th Conference on*, pages 7282–7287. IEEE, 2016. `doi:10.1109/CDC.2016.7799393`.

27 Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, 2014. `doi:10.1145/2619998`.

28 Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys*, 47(4), May 2015. `doi:10.1145/2716320`.

29 André DeHon, Jean-Louis Giavitto, and Frédéric Gruau, editors. *Computing Media and Languages for Space-Oriented Computation*, volume 06361 of *Dagstuhl Seminar Proceedings*, 2007. URL: `http://drops.dagstuhl.de/portals/06361`.

30 Sébastien Doeraene. Cross-platform language design in Scala.js (keynote). In *SCALA@ICFP*, page 1. ACM, 2018. `doi:10.1145/3241653.3266230`.

31 Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), June 2003. `doi:10.1145/857076.857078`.

32 Alois Ferscha. Collective adaptive systems. In *UbiComp/ISWC Adjunct*, pages 893–895. ACM, 2015. `doi:10.1145/2800835.2809508`.

33 Debasish Ghosh, Justin Sheehy, Kresten Krab Thorup, and Steve Vinoski. Programming language impact on the development of distributed systems. *J. Internet Serv. Appl.*, 3(1):23–30, 2012. `doi:10.1007/s13174-011-0042-y`.

34 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. `doi:10.1145/503502.503505`.

35 Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005. doi:10.1145/1082469.1082470.

36 Pushpendu Kar, Arijit Roy, and Sudip Misra. Connectivity reestablishment in self-organizing sensor networks with dumb nodes. *ACM Trans. Auton. Adapt. Syst.*, 10(4):28:1–28:30, 2016. doi:10.1145/2816820.

37 Naomi Kuze, Daichi Kominami, Kenji Kashima, Tomoaki Hashimoto, and Masayuki Murata. Hierarchical optimal control method for controlling large-scale self-organizing networks. *ACM Trans. Auton. Adapt. Syst.*, 12(4):22:1–22:23, 2018. doi:10.1145/3124644.

38 Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. Asynchronous distributed execution of fixpoint-based computational fields. *Log. Methods Comput. Sci.*, 13(1), 2017. doi:10.23638/LMCS-13(1:13)2017.

39 Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87–119, 1994. doi:10.1145/174666.174668.

40 Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Pervasive Computing and Communications, 2004*, pages 263– 273. IEEE, 2004. doi:10.1109/PERCOM.2004.1276864.

41 Yuanqiu Mo, Jacob Beal, and Soura Dasgupta. An aggregate computing approach to self-stabilizing leader election. In *International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 112–117. IEEE, 2018. doi:10.1109/FAS-W.2018.00034.

42 Christine Morin and Isabelle Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Trans. Parallel Distributed Syst.*, 8(9):959–969, 1997. doi:10.1109/71.615441.

43 Hamid Mushtaq, Zaid Al-Ars, and Koen Bertels. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In *International Design and Test Workshop (IDT)*, pages 12–17. IEEE, 2011. doi:10.1109/IDT.2011.6123094.

44 Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks*, pages 78–87, 2004. doi:10.1145/1052199.1052213.

45 Torsten Hoefler on behalf of the MPI Forum. MPI: A message-passing interface standard, version 2.2. Specification, Message Passing Interface Forum, September 2009. URL: http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

46 H. Van Dyke Parunak, Sven Brueckner, Robert S. Matthews, and John A. Sauter. Pheromone learning for self-organizing agents. *IEEE Trans. Syst. Man Cybern. Part A*, 35(3):316–326, 2005. doi:10.1109/TSMCA.2005.846408.

47 Rajiv Ranjan, Omer F. Rana, Surya Nepal, Mazin Yousif, Philip James, Zhenya Wen, Stuart L. Barr, Paul Watson, Prem Prakash Jayaraman, Dimitrios Georgakopoulos, Massimo Villari, Maria Fazio, Saurabh Kumar Garg, Rajkumar Buyya, Lizhe Wang, Albert Y. Zomaya, and Schahram Dustdar. The next grand challenges: Integrating the internet of things and data science. *IEEE Cloud Comput.*, 5(3):12–26, 2018. doi:10.1109/MCC.2018.032591612.

48 Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Engineering resilient collective adaptive systems by self-stabilisation. *ACM Trans. Model. Comput. Simul.*, 28(2):16:1–16:28, 2018. doi:10.1145/3177774.

49 Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.*, 109, 2019. doi:10.1016/j.jlamp.2019.100486.

50 Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. Distributed system development with ScalaLoci. *Proc. ACM Program. Lang.*, 2(OOPSLA):129:1–129:30, 2018. doi:10.1145/3276499.