The eXchange Calculus (XC): A functional programming language design for distributed collective systems

Giorgio Audrito a,∗, Roberto Casadei b, Ferruccio Damiani a, Guido Salvaneschi c, Mirko Viroli b

a Dipartimento di Informatica, University of Turin, Turin, Italy
b DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Italy
c Institute of Computer Science, University of St.Gallen, St.Gallen, Switzerland

ARTICLE INFO

Keywords:
Distributed programming
Collective computing
Core calculus
Operational semantics
Type soundness
Scala DSL
C++ DSL

ABSTRACT

Distributed collective systems are systems formed by homogeneous dynamic collections of devices acting in a shared environment to pursue a joint task or goal. Typical applications emerge in the context of wireless sensor networks, robot swarms, groups of wearable-augmented people, and computing infrastructures. Programming such systems is notoriously hard, due to requirements of scalability, concurrency, faults, and difficulty in making desired collective behaviour ultimately emerge: ad-hoc languages and mechanisms have been proposed threads like spatial computing, macro-programming, and field-based coordination.

In this paper we present the eXchange Calculus (XC), formalising a tiny set of key mechanisms, usable across many different languages and platforms, allowing to express the overall interactive behaviour of distributed collective systems in a declarative way. In this approach, computation (executed in asynchronous rounds), communication (which is neighbour-based), and state over time, are all expressed by a single declarative construct, called exchange. We provide a formalisation of XC in terms of syntax, device-level and network-level semantics, prove a number of properties of the calculus, and discuss applicability considering a smart city scenario. XC is implemented as a DSL in Scala and in C++, with different trade-offs in terms of productivity and platform targeting.

1. Introduction

Research trends like the Internet of Things (Atzori et al., 2010) promote a vision of large-scale deployments of devices capable of computation, communication, and physical interaction with the environment. A particular kind of system is what we call a distributed collective system, namely one consisting of a roughly homogeneous collection of devices cooperating with neighbours to pursue common goals. Notable examples include wireless sensor networks (WSNs) (Mottola and Picco, 2011), swarms of robots (Brambilla et al., 2013), groups of wearable-augmented people (Abowd, 2016), and computing ecosystems (Pianini et al., 2021b). Despite possible differences in capabilities, these groups of individuals are reasonably similar, and hence form a collective rather than a composite (Masolo et al., 2020). Building applications that fully exploit the potential of such distributed systems is a matter of supporting intelligent behaviour not just at the individual level, but also at the system or collective level (Tumer and Wolpert, 2004; Nicola et al., 2020). The goal may be addressed by automated approaches like multi-agent reinforcement learning (Zhang et al., 2019), or by language-based, programming approaches sometimes referred to as macro-programming (Casadei, 2023; Newton and Weish, 2004; Sene Júnior et al., 2022), that notably include spatial computing (De-Hon et al., 2007), field-based computing (Viroli et al., 2019; Lluch-Lafuente et al., 2017; Mamei and Zambonelli, 2004), ensemble-based programming (De Nicola et al., 2014; Abd Alrahman et al., 2020).

Inspired by such research, in this work we describe a novel programming language design, called XC, aimed to support the development of collective adaptive behaviour while abstracting the management of low-level aspects like concurrency, asynchronous execution, communication, and failure. In our approach, the programmer develops a single, integrated program that represents the overall collective task, by including the control logic of each individual or sub-group thereof. Then, the intended collective behaviour emerges in a self-organising way by the repeated local execution of sense–compute–interact rounds by all the devices. The language design, generalising over field calculi (Viroli et al., 2019), is derived from the typed lambda calculus and based on...
a single communication built-in, called exchange, capturing both state and communication management, and also allowing differentiated messages to be sent to neighbours, for increased practical expressiveness. Thanks to a mechanism that we call alignment, programs are written in a way that retain composability, hence making it possible to devise and compose functional blocks of collective adaptive behaviour.

To summarise, this paper provides the following contributions:

1. We describe the design of XC, a programming language for distributed collective systems that abstracts over concurrency, network communication, message loss, and device failures. Crucially, XC retains compositionality even with asynchronous communication, thanks to alignment.

2. 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-organising systems (Mills, 2007).

3. We provide a formalisation of a core calculus for XC, including syntax and operational semantics and type system. We prove the type soundness and round determinism property, leveraging the co-inductive approach proposed in Ancona et al. (2017b), and discuss the automatic importing in XC, via subsumption of the field calculus (Viroli et al., 2018), of robustness and expressiveness results.

4. We implement XC as publicly available Scala and C++ internal domain-specific languages (DSLs), together targeting a number of different execution platforms.

5. 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.

This manuscript is an extended version of the conference paper (Audrito et al., 2022a), where the additional contributions, mainly related to the formalisation, include the parts on typing (Section 4.2), network semantics (Section 4.3), and formal proofs regarding types (Section 5.1), expressiveness (Section 5.2), and self-stabilisation (Section 5.3).

The paper is structured as follows. Section 2 introduces XC design. Section 3 demonstrates XC through examples. Section 4 presents a formalisation of XC. Section 5 provides formal properties of XC. Section 6 discusses the implementation. Section 7 evaluates XC. Section 8 compares XC to various threads of related work. Section 9 concludes and outlines future research directions.

2. XC language design

In this section, we describe the design of XC by a high-level perspective. In particular, we present the key elements of the programming model: the basic system model and its assumptions (Section 2.1), the data structure of neighbouring values (Section 2.2), the only communication primitive, exchange (Section 2.3), compositionality as the major benefit of the programming model (Section 2.4), conditionals for disjoint sub-computations (Section 2.5), and a preview of the benefits in terms of inherent fault tolerance (Section 2.6).

2.1. System model

Asynchronous, round-based execution and communication. Our target systems are modelled as a collection of devices, generally equipped with sensors and actuators, that repeatedly compute the same program and communicate asynchronously with neighbours by exchanging messages. The neighbour set of any device is dynamic: it can change dynamically, e.g., as a result of mobility, failure, and network delay. Device behaviour is modelled through a notion of (execution) round, whereby a device independently “fires”, gathers local context (sensor data and messages from neighbours), “atomically executes” a XC program, and then acts on the local context as prescribed by the program. Executing a program results in the production of an output (the program’s return value), which may be used to describe actions, as well as, implicitly, the messages that have to be sent to neighbours for coordination purposes, before waiting to execute the round again—sometimes we say a device “wakes up”, executes the round, and then “goes back to sleep”. As mentioned, the behaviour of each device in the network is developed as a single program.1 Such rounds of execution may be scheduled 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 may run out of battery and never wake up again, or it can restart after a long time if the battery gets charged. Therefore, rounds—and hence the communication among devices—are entirely asynchronous.

Last-message buffering and dropping. The messages received by a sleeping device are collected in a buffer where only the most recent message per neighbour is kept and where messages whose exceed a certain configurable lifetime are dropped (expiration). 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 Fig. 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, $e_1$ and $e_2$. Grey arrows denote messages that get lost and are never received. The computation $e_2$ 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 $\delta_1$ and to $\delta_4$. The order of messages from a same sender is preserved but, other than that, there are very few assumptions on messages. If a device $\delta_i$ runs multiple rounds before a device $\delta_j$ even runs a single round, $\delta_i$ sees only the message received from the last round of $\delta_i$, 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

1 This approach is often referred to as macroprogramming (Casadei, 2023; Newton and Welsh, 2004) or multi-tier programming (Weizenburger et al., 2020, 2018). Notice that it does not limit possible behaviours as devices can still exhibit different executions of the same program.
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 de-emphasize 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 (Ahmad et al., 1995). 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 $\epsilon$ include traditional types $A$ like float, string or list. Neighbouring values (nvalues) are a map $\nu$ from device identifiers $\delta$ to corresponding local values $\epsilon_{\delta}$, with a default $\epsilon$, written $\epsilon[\delta_1 \rightarrow \epsilon_1, \ldots, \delta_n \rightarrow \epsilon_n]$. 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 $\nu$ is everywhere (i.e., for all neighbours) except for devices $\delta_1, \ldots, \delta_n$ with values $\epsilon_1, \ldots, \epsilon_n$”. To exemplify nvalues, in Fig. 1, upon waking up for computation $\delta_2$, $\delta_3$ may process a nvalue $\nu = 0[\delta_1 \rightarrow 1, \delta_2 \rightarrow 2, \delta_3 \rightarrow 3]$, corresponding to the messages carrying the scalar values $1, 2, 3$ and received when asleep from $\delta_1, \delta_3$, 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 $\nu' = 0[\delta_1 \rightarrow 5, \delta_2 \rightarrow 6]$; so that 5 is sent to $\delta_1$, 6 is sent to $\delta_2$, and 0 is sent to every other device (such as a newly-connected device with no dedicated value yet). We also use the notation $\nu'(\delta')$ for the local value $\epsilon'$ if $\delta' \rightarrow \epsilon'$ is in $\nu$, or the default local value $\epsilon'$ of $\nu'$ otherwise, reflecting the interpretation of nvalues as maps with a default. For instance, $\nu'(\delta_1) = 6$ and $\nu'(\delta_2) = 0$. To help the reader, in code snippets, we underline the variables holding neighbouring values, and, similarly, we underline a primitive type $A$ to indicate the type of an nvalue $\nu = \epsilon[\delta_1 \rightarrow \epsilon_1, \ldots, \delta_n \rightarrow \epsilon_n]$ where $\epsilon_1, \ldots, \epsilon_n$ have type $A$.

**Nvalues generalise local values.** A local value $\epsilon$ can be automatically converted to a nvalue $\epsilon[\delta]$ 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 a nvalues where all devices are mapped to a default value. In the formalisation (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 $\nu = 0[\delta_1 \rightarrow 1, \delta_1 \rightarrow 2]$ and $\nu = 1[\delta_1 \rightarrow 2]$, we have $\nu' = \nu + \nu = 1[\delta_1 \rightarrow 3, \delta_1 \rightarrow 3]$. Note that $\delta_1 \rightarrow 3$ in $\nu'$ is due to the fact that $\delta_1 \rightarrow 2$ in $\nu$, and $\delta_1$ has default value 1 in $\nu$. Using also the automatic promotion of local values to nvalues, we have that $\nu + 1 = 0[\delta_1 \rightarrow 1, \delta_1 \rightarrow 2] + 1 = 1[\delta_1 \rightarrow 2, \delta_1 \rightarrow 3]$.

**Operations on nvalues.** Besides pointwise manipulation, nvalues can be folded over, similar to a list, through built-in function $\text{nfold}(f : (A, B) \rightarrow A, \nu : B, \epsilon : A)$, where the function $f$ is repeatedly applied to neighbours’ values in a field $\nu$ (thus excluding the value for the self device), starting from a base local value $\epsilon$. For instance, assume that $\delta_1$ is performing a $\text{nfold}$ operation, with the current set of neighbours $\{\delta_1, \delta_3\}$. Then $\text{nfold}(+, \nu_1, 10) = 10 + \nu_1(\delta_1) + \nu_1(\delta_3) = 10 + 2 + 1$, where $\nu_1(\delta_1) = 1$, $\delta_1 \rightarrow 1$, and $\delta_3 \rightarrow 2$ 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 $\delta$ provided by its neighbours. To this end, one selects the minimum (using $\text{nfold}$ with starting value $\text{Infinity}$) of neighbours’ estimates $\delta$ increased by the relative distance estimates $\text{senseDist}$ (provided by a sensor in the device).

```plaintext
1 def distanceEstimate(\delta) { // type: (num) -> num
2     return \text{nfold}(\text{min}, \delta + \text{senseDist}, \text{Infinity})
3 }
```

Notice that $\delta$ and $\text{senseDist}$ sum up neighbour-wise; if neighbour $\delta$ shares estimate $\delta(\delta)$, the node’s best estimate from that neighbour is $\delta(\delta) + \text{senseDist}(\delta)$. The minimum among all estimates (and $\text{Infinity}$) is selected.

Additional built-in operations on nvalues are $\text{self}(\nu : A) : A$, which returns the local value $\nu(\text{self})$ in $\nu$ for the self device $\delta$, and $\text{updateSelf}(\nu : A, \epsilon : A) : A$ which returns a nvalue equal to $\nu$ except for the self device $\delta$ updated to $\epsilon$. The substitution notation stand for default mapped updates, so that $\text{updateSelf}(\nu, \epsilon) = \nu[\delta \rightarrow \epsilon]$. Indeed, the notation $\nu[\delta_1 \rightarrow \epsilon_1, \ldots]$ for nvalues can be understood as a substitution updating $\epsilon$ (the map equal to $\epsilon$ everywhere) by associating $\epsilon_n$ 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 tolerant changes of neighbourhoods between rounds. Yet, non-uniform behaviour can be encoded via built-in function $\text{uid}$ (combined with communication primitives, Section 2.3), which provides the unique identifier $\delta$ of the current device.

Fig. 2 shows a summary of every built-in function used in this paper. Constructors and point-wise operators are standard; the multiplexer operator $\text{mux}(\epsilon_1, \epsilon_2, \epsilon_3)$ returns $\epsilon_i$ if $\epsilon_i$ is True, $\epsilon_3$ otherwise. We also omit pair and use the shortcut $(v_1, v_2)$ 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

\[
\text{exchange}(e_i, e_j) = \text{return } e_i, \text{ send } e_j
\]

which de-sugars to

\[
\text{exchange}(e_i, e_j) = (e_i, e_j)
\]

and is evaluated as follows. (i) The device computes the local value \( e_i \) of \( e_i \) (the initial value). (ii) It substitutes variable \( e_i \) with the nvalue \( v \) of messages received from the neighbours for this exchange, using \( e_i \) as default. The exchange returns the (neighbouring or local) value \( v \) from the evaluation of \( e_j \). (iii) \( e_j \) evaluates to a nvalue \( v_i \) consisting of local values to be sent to neighbour devices \( \delta \), that will use their corresponding \( v_i \) (\( \delta \)) as soon as they wake up and perform their next execution round.

Often, expressions \( e_i \) and \( e_j \) coincide, hence we provide

\[
\text{exchange}(e_i) = \text{return } e_i, \text{ send } e_i
\]

as a shorthand for

\[
\text{exchange}(e_i) = (e_i, e_i)
\]

Another common pattern is to access neighbours' values, which we support via\( \text{nbr}(e_i, e_j) = \text{exchange}(e_i, e_j) = \text{return } e_i, \text{ send } e_j \). In \( \text{nbr}(e_i, e_j) \), the value of expression \( e_j \) is sent to neighbours, and the values received from them (gathered in \( \text{nbr} \) together with the default from \( e_i \)) are returned as a nvalue, thus providing a view on neighbours' values of \( e_i \).

It is crucial for the expressiveness of XC that exchange (hence \( \text{nbr} \)) 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", which are a reusable data type for every neighbour to the number of times a mutual connection has been established with it.

\[
\text{def ping-pong}() \{ \text{// type: ()} \to \text{num} \\
\text{exchange( 0, } (z) \to \text{retrans} z + 1 \}
\]

Every time a device evaluates ping-pong, it first gathers a neighbouring value \( z \) associating neighbours to their respective connection counter – 0 is for newly connected devices. Expression \( z + 1 \) is computed substituting \( z \) for \( z \) incrementing each such counter (including for newly connected devices, which now map to 1). The resulting value \( z + 1 \) 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 breaks and the corresponding messages expire, the connection counter reverts 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 to a set of source devices results in a gradient (Audrito et al., 2017a). 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 (Dasgupta and Beal, 2016).

\[
\text{def distanceTo}(src) \{ \text{// type: (bool) \to num} \\
\text{exchange( Infinity, } (z) \to \text{retrans} \text{mux}(src, 0, \text{distanceEstimate}(z)) \}
\]

Its repeated application in a 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 \text{mux} (i.e., a strict version of \text{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 function call \text{distanceEstimate} (thus sharing their value \( z \)). Any change in the network (e.g., due to failure, mobility, dynamic joining) directly affects the domain of \( z \), hence 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 by \text{alignment} that 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:

\[
\text{def average}(weight, value) \{ \text{// type: (num) \to num} \\
\text{val totV = nfld(+, nbf(0, weight, weight));} \\
\text{val totW = nfld(+, nbf(0, weight*value), weight*value));} \\
\text{totV / totW}
\]

The Journal of Systems & Software 210 (2024) 111976
First, the total weight of neighbours is computed in Line 2, by first producing a nvalue of neighbours’ weights through \( \text{nbr}(0, \text{weight}) \), and then reducing it to its total by \( \text{nfold} \), 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 \(*\text{value}\) of neighbours (including the current device) are added. The weighted average is then obtained and returned by the function as \( \text{totV/totW} \).

This function contains two calls to \( \text{nbr} \), in turn calling the \( \text{exchange} \) built-in, both with messages of type \( \text{num} \). XC ensures that the messages from the different communicating routines are correctly dispatched to neighbours, each used only in the corresponding call to \( \text{exchange} \) 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. Fig. 3 demonstrates alignment. Top-left is a tree representation of the XC program in Example 4, accounting for stack frames and children of 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 \( \text{closestFire} \) 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.

```plaintext
1 def closestFire(temperature, smoke) { //type: (bool, bool) → num
2     (num, num) → num
3     val trust = nfold(+, 1, 1);
4     val hot = average(trust, temperature) > 60;
5     val cloudy = average(trust, smoke) > 10;
6     distanceTo(hot and cloudy)
7     }
```

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 \( \text{nfold}(+, 1, 1) \). 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 \( \sum \) of received messages in an exchange, or for determining which values of an 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 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.

```plaintext
1 // type: (bool, bool) → num
2 define distanceToGateways(local, gateway) {
3     if (local) { Infinity } else { distanceTo(gateway) }
4 }
```
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-locals.

Now suppose we would like the local subgroup to compute distances from local requesters, and the other subgroup to still compute distances from gateways, always excluding the devices of the complementary group.

```plaintext
// type: (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, messages exchanged by devices in the local group must not be matched with those exchanged by device outside the group, otherwise every device would just compute their distance from the closest local requester or non-local gateway, which is not the intended behaviour. XC grants that this does not happen, as exchange expressions arising from different branches have different stack frames, hence happen in separate interaction domains.

Fig. 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. XC Design: Discussion

The model and abstractions presented in this section encourage developers to build collective behaviours (as XC programs) as compositions of functional blocks that are resilient to failures. In other words, by its very design, XC enjoys the following features.

Automatic failure absorption. In the 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 inherent fault tolerance and resilience with respect to devices that dynamically join and leave the set of neighbours (e.g., because they physically change location), transparently from programmers. 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.

Compositionality and collective stance. Programming resilient collective behaviour is supported by functional abstraction and composition: simpler resilient blocks of collective behaviour (cf. gradients in Example 3) can be defined (def) as reusable functions and composed together to build more complex applications, while retaining properties like fault tolerance and self-stabilisation (cf. Section 5). More examples of this feature are provided in Section 3 and in the case study of Section 7. This feature is also due to the fact that each behaviour can be described as a function that encapsulates both the processing and the communication required to come up with a coordinated result, while the round-based execution model supports progress and incorporation of local changes (ultimately propagating from neighbourhoods to neighbourhoods up to the global system).
3. XC at Work

We now show XC in action by means of example applications in areas like WSNs, Internet of Things (IoT), and large-scale cyber-physical systems (CPSs). 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 7).

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.

```python
1 def gossipEver(event) { // type: (bool) → bool
2     exchange(False, (n) => resend
3         nfold(or, n, self(n) or event))
4 }
```

The first argument of the exchange (Line 3) sets the initial value to False for n (and thus for newly-connected devices, including for the current device in its first round). The second argument is a lambda, whose parameter n is the nvalue representing the gossips of neighbouring devices (including the current device itself, for which n includes the gossip value in its previous round). Function nfold collapses the neighbours’ gossips through binary operation or (checking whether there is any true gossip), with the starting value self(n) or event which holds 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 the function and sent to each neighbour.

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).

```python
1 def broadcast(dist, value, null) { // type: (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 }
```

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 g (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, so that nbr(False, parent) is true. Every other neighbour receives null (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.

```python
1 // type: (bool, bool, num.A,A,A) → A
2 def channelBroadcast(source, dest, width, value, null) {
3     val ds = distanceTo(source);
4     val dd = distanceTo(dest);
5     val sd = broadcast(ds, dd, Infinity);
6     val channel = ds + dd <= sd + width;
7     if (channel) { broadcast(ds, value, null) } else { null }
8 }
```

The channel region is computed through the geometrical definition of ellipse (Line 6): 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 7).

The example illustrates functional composition: channelBroadcast composes multiple instances of distanceTo (Example 3) and broadcast (Example 8) to realise a more complex behaviour. The composition preserves the self-organising properties of its constituent parts, being able to automatically adapt to failures, mobility and changes in source, dest, width.

So far, we presented functions building a communication structure to disseminate information over the network. Yet, we have not addressed the problem of collecting such information, especially in the non-trivial case where it is obtained by inspecting the whole network.
Example 10 (Information Collection). The collect algorithm (inspired by Audrito et al., 2021a) 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.

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

```java
1 def accumulate(v, w) { // type: (A num) --> A
2   (A,A) --> A
3   (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:

```java
1 def accumulate(v, w) { // type: (A num) --> A
2   (A,A) --> A
3   (A,num) --> A
```

Improvements over (Audrito et al., 2021a) are both stylistic (cleaner code) and in the precision of weights, since in Audrito et al. (2021a) 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 operation 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—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 operation centres. This logic is implemented by function smartC (Fig. 5), which reuses distanceToChannels from detectors to the closest operations centres (Examples 3 and 8 to 10).

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 2). The nodes that are inspected are only those for which the gradient value is less than inspectionRadius (Line 3). Then, two different behaviours are defined based on whether a node is inspected or not (Line 5). Nodes not inspected just return nullReport (Lines 4 and 19). In the domain of inspected nodes, including the detector, a collection process is activated (Line 7 to 11) 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 12) and decide whether the warning level is high (Line 13): such a decision (warning significance) is broadcast from the detector to the rest of the area (Line 14), as a kind of notification to the devices in the surroundings. Also, depending on whether the warning level is high (Line 15 to 17), it either collects the logs from all the nodes in the area (Line 16), 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 20).

```java
1 def smartC(isDetector, isOpsCentre, channelWidth, inspectionRadius, commRadius, localWarning, warningThreshold, localLog, nullLog, logCat) {
2   val dist = distanceToChannels;
3   val inspected = dist < inspectionRadius;
4   val nullReport = (uid, 0, nullLog);
5   val report = if (inspected) {
6     val (numWarning, numNodes) = collect(detDist, commRadius, localWarning, 1.0,)
7     (v, l) => (nfold(+, log(v), l))
8   } else {
9     nullReport
10   }
11   val meanWarning = sumWarning / numNodes;
12   val localWarning = meanWarning > warningThreshold;
13   val logs = if (warning) {
14     collect(detDist, commRadius, localLog, logCat, {v,l}=>v)
15   } else {
16     nullLog
17   }
18   channelBroadcast(isDetector, isOpsCentre, channelWidth, report, nullReport)
19 }
```

Fig. 5. XC implementation of a smart city monitoring application.
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 (Audrito et al., 2018a), with functional compositionality of global behaviour.

The formalisation of typing (Section 4.2) and of device-level semantics (first part of Section 4.3) is inspired by classical functional languages with let-polymorphism like ML, while differing in crucial ways: in the presence and handling of nvalues, and in the alignment-based semantics. Due to these significant differences, ML soundness 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 (Audrito et al., 2018a), with functional compositionality of global behaviour.

The formalisation of typing (Section 4.2) and of device-level semantics (first part of Section 4.3) is inspired by classical functional languages with let-polymorphism like ML, while differing in crucial ways: in the presence and handling of nvalues, and in the alignment-based semantics. Due to these significant differences, ML soundness 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 (Audrito et al., 2018a), with functional compositionality of global behaviour.

The formalisation of typing (Section 4.2) and of device-level semantics (first part of Section 4.3) is inspired by classical functional languages with let-polymorphism like ML, while differing in crucial ways: in the presence and handling of nvalues, and in the alignment-based semantics. Due to these significant differences, ML soundness 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 (Audrito et al., 2018a), with functional compositionality of global behaviour.

4.1. Syntax

Fig. 6 (top) shows the syntax of FXC. As in Igarashi et al. (2001), 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 \( e \) can be either:

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

The set of built-in functions includes sensors \( s \), the communication built-in \( \text{exchange} \), functions manipulating nvalues, and further built-ins as needed (e.g., for numbers, pairs, lists, etc.). 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 4.3, 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 (Fig. 6, middle), and an expression \( e \) is closed if \( \text{dom}(e) = \emptyset \).

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 Fig. 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 Fig. 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 \( \text{mux} \) operator and then applying it.

4.2. Typing

Fig. 7 presents a classic Hindley-Milner type system (Damas and Milner, 1982) for FXC. A type \( T \) can be:

- a type variable \( a \);
- a (recursive) data type \( K(\overline{T}) \), consisting of a parametric type name \( K \) of arity \( n \geq 0 \), applied to \( n \) types \( T_1, \ldots, T_n \) (possibly zero);
- or a function type \( (\overline{T}) \rightarrow T \).

Since local values and nvalues are treated uniformly in the FXC semantics, and nvalues are the only values produced by evaluating (closed) expressions, there is no need for dedicated types for nvalues: an nvalue gets the type of its messages. Indeed, the underline notation to highlight nvalues types and variables is only for the readers’ convenience and does not need to be modelled in the type system. We write \( TV(T) \) for the set of type variables in \( T \). Polymorphism of functions and data constructors is supported by type schemes \( T \cdot S \) of the form \( \forall \overline{\pi}. T \) where \( \pi \) occur free in \( T \), representing all types obtained by substituting \( \overline{\pi} \) with types \( \overline{\pi} \), as per the type scheme instantiation relation \( < \). A typing environment \( A \) is a set of assumptions \( a : T \cdot S \) where the assumption subject \( a \) can be either a variable, a built-in function or constructor.

We assume that sensors \( s \) have types \( s : T \) where \( TV(T) = \emptyset \). In the typing of programs, an initial typing environment \( A_0 \) declares a (unique) type scheme for every available constructor and built-in function. This \( A_0 \) is then extended with (unique) assumptions for bounded variables encountered while typing sub-expressions of the program. We write \( A(a) \) for the unique type scheme in \( A \).

We specify typing of expressions via judgements \( A \vdash e : T \) which read ‘expression \( e \) has type \( T \) under assumptions \( A \)’. Following Igarashi et al. (2001), multiple overbars are expanded together (e.g., \( A \vdash \overline{\pi} : \overline{T} \) stands for \( A \vdash e_1 : T_{i1}, \ldots, A \vdash e_n : T_{in} \)). Typing judgements for expressions are syntax-directed. All rules are standard except \( \text{V-NVAL} \), which ensures that all messages \( \ell, \overline{\pi} \) of a nvalue have a same type \( T \), and there are no repetitions in \( \overline{\pi} \).
4.3. Semantics

The semantics of FXC expressions is defined in terms of:

1. an operational big-step device semantics, formalising the computation of a device within one round; and
2. a denotational network semantics, formalising how the computation of devices give rise to the overall network evolution.

Device-level semantics. Fig. 8 presents the device semantics, formalised by judgement \( \delta; \sigma; \Theta \vdash e \Downarrow \upsilon; \theta \), to be read as “expression \( e \) evaluates to nvalue \( \upsilon \) and value-tree \( \theta \) on device \( \delta \) with respect to sensor values \( \sigma \) and value-tree environment \( \Theta \)”.

- \( \upsilon \) is called the result of \( e \);
- \( \theta \) is an ordered tree with nvalues on some nodes (cf. Fig. 8 (top)), representing messages to be sent to neighbours by tracking the nvalues produced by exchange-expressions in \( e \), and the stack frames of function calls;
- \( \sigma \) collects the (non expired) value-trees received by the most recent firings of neighbours of \( \delta \), as a map \( \delta_1 \rightarrow \theta_1, ..., \delta_n \rightarrow \theta_n \) from device identifiers to value-trees;
- \( \Theta \) associates nvalues to every sensor name \( \mathbf{s} \) and in case of open expression, to every free variable \( \mathbf{x} \) occurring in \( e \).

The semantics is based on value-trees as means of communication, as they allow grouping messages relative to every exchange construct executed in a single round, while storing alignment information within their tree structure by encoding all stack frames encountered during the execution.

The syntax of value-trees and value-tree environments is in Fig. 8 (top). The rules for judgement \( \delta; \sigma; \Theta \vdash e \Downarrow \upsilon; \theta \) (Fig. 8, middle) defines the semantics also for open expressions is convenient to enable induction in the upcoming proofs.
are standard for functional languages, extended to evaluate a sub-expression \( \text{e}' \) of \( \text{e} \) 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 \( \Theta \). This alignment process exploits the auxiliary “projection” functions \( \pi_i \) for any positive natural number \( i \) (Fig. 8, top). When applied to a value-tree \( \delta \) in \( \Theta \), \( \pi_i \) returns the \( i \)th sub-tree \( \delta_i \) 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 value-trees in 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 \( \text{f}((\text{e}_1, \ldots, \text{e}_n)) \), the environment \( \Theta \) is reduced to the smaller set \( \Theta' \) of trees which corresponded to the evaluation of a function with the same name (as defined in Fig. 8 (top)).

Rule \([E\text{-VAL}]\) evaluates an nvalue \( \nu \) to \( \nu \) itself and the empty value-tree. Rule \([E\text{-LIT}]\) evaluates a local literal \( \text{e} \) to the nvalue \( \text{e}[\nu] \) and the empty value-tree. Rule \([E\text{-VAR}]\) evaluates a free variable \( \text{x} \) to the nvalue stored for it in \( \sigma \). Rule \([E\text{-VAL}]\) evaluates a val-expression, by evaluating the first sub-expression with respect to the first sub-tree of the environment obtaining a result \( \nu_1 \), and then the second sub-expression with respect to the second sub-tree of the environment, after substituting the variable \( \text{x} \) with \( \nu_1 \).

Rule \([E\text{-APP}]\) is mostly standard eager function application: the function expression \( \text{e}_0 \) and each argument \( \text{e}_i \) are evaluated w.r.t. \( \pi_{i+1}(\Theta) \) producing result \( \nu_i \) and value-tree \( \Theta_i \). Then, the function application itself is demanded to the auxiliary evaluation rules, w.r.t. the last subtree of the trees corresponding to the same function name \( \pi_{n+2}(\Theta) \) during the auxiliary rule \([A\text{-FUN}]\) handles the application of fun-expression, which evaluates the body after replacing the arguments \( \Sigma \) with their provided values \( \Pi \), and the function name \( \text{x} \) with the fun-expression itself. Rules \([A\text{-UID}]\) and \([A\text{-SELF}]\) trivially encode the behaviour of the \( \text{uid} \) and \( \text{self} \) built-ins. Rule \([A\text{-SENS}]\) defines the behaviour of sensors through the values stored in \( \sigma \). Rule \([A\text{-XC}]\) evaluates an exchange-expression, realising the behaviour described at the beginning of Section 2.3. Notation \( \nu_1[\delta \mapsto \nu(\delta)] \) is used to represent the nvalue \( \nu_1 \) after the update for each \( i \) of the message for \( \delta_i \) with the custom message \( \nu(\delta) \). The result is fed as argument to function \( \nu \): 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 \([A\text{-FOLD}]\) encodes the \( \text{nfold} \) operators. First, the domain of \( \Theta \) is inspected, giving a (sorted) list \( \delta_0 \). An initial local value \( \nu_0 \) is set to the “self-message” of the third argument. Then, a sequence of \( \mathcal{\tilde{e}} \) is defined, each by applying function \( \nu_0 \) to the previous element in the sequence and the value \( \nu_1(\delta_i) \) (skipping \( \delta \) itself). The final result \( \mathcal{\tilde{e}} \) 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-level semantics.** The evolution of a network executing a program \( \text{e}_{\text{main}} \) is formalised through structures of events, atomic rounds of computation performed by devices according to the device-level semantics. Such events across space (i.e., the devices where they happen) and across time (i.e., when they happen w.r.t. other events) can be seen as the execution of the program on a single “aggregate machine” (Beal et al., 2015) in which information is exchanged across events (and hence, devices) following a messaging relationship \( \rightarrow \). This idea can be formalised through augmented event structures, which augment a classical event structure (Lamport, 1978) with further information associated to events: device identifiers and sensor status.

**Definition 1 (Augmented Event Structure).** An augmented event structure \( E = (\mathcal{\mathcal{E}}, \rightarrow, d, s) \) is a tuple where:

- \( d : E \rightarrow \Delta \) is a mapping from events to the devices where they happened, and
- \( s : E \rightarrow S \) maps each event \( e \) to a sensors status \( s \) (as in the device-level semantics),

such that:

- \( e_1 = e_2 \) whenever \( d(e_1) = d(e_2) \) and \( e_1 \rightarrow e \) for \( i = 1, 2 \) (i.e., all predecessors of an event happened on different devices);
- there are no sequences \( e_1 \rightarrow \ldots \rightarrow e_i \) (i.e., the \( \rightarrow \) relation is acyclic);
- the set \( \mathcal{\mathcal{E}} = \{ e' \in E \mid e' \rightarrow \ldots \rightarrow e \} \) of events that can reach \( e \) in \( \rightarrow \) is finite for all \( e \) (i.e., \( \rightarrow \) is well-founded and locally finite).

We say that event \( e' \) is a supplier of event \( e \) iff \( e' \rightarrow e \). We call causality relation the irreflexive partial order \( < \subseteq E \times E \) obtained as the transitive closure of \( \rightarrow \).

An example of augmented event structure is in Fig. 9 (top left). The example is based on the evolving topology in Fig. 9 (top right). It illustrates how the causality relation effectively partitions the set of events, w.r.t. a reference event \( e \), into sub-spaces: the “causal past”, “causal future”, and “present” (concurrent). A computation at \( e \) can use the information from past events, and its outcomes can potentially affect any future event. The messaging relation induces causality by expressing when an event can directly influence another (by sending a message).

This computational model captures the behaviour of communicating physical devices. The evolution of the state of devices is a chain of \( \rightarrow \)-connected events associated to the same device. Communication is captured by \( \rightarrow \)-connected events associated to different devices. Such a computational model allows us to express programs abstracting from synchronisation, shared clocks, or regularity and frequency of events. Informally, following previous work (Mamei and Zambonelli, 2004; Lluch-Lafuente et al., 2017; Viroli et al., 2018), we refer to a field of values as a mapping from devices to values, capturing a global snapshot of the values produced by the most recent firing of each device. The evolution over time of a field of values is a global data structure, a space-time value, which maps each event (in an augmented event structure) to a corresponding value.

**Definition 2 (Space-Time Value).** Let \( E = (\mathcal{\mathcal{E}}, \rightarrow, d, s) \) be an augmented event structure and \( \mathcal{\mathcal{V}}_T \) be the domain of nvalues of type \( T \). A space-time value (in \( E \) of type \( T \)) is a function mapping events to nvalues \( \Phi : E \rightarrow \mathcal{\mathcal{V}}_T \).

In this computational model, the evaluation of a program \( \text{e}_{\text{main}} \) in an augmented event structure \( \mathcal{\mathcal{E}} \) then produces a space-time value, induced by repeatedly applying the device-level semantics.

**Definition 3 (Program Evaluation on Event Structures).** Let \( e \) be an FXC expression of type \( T \) given assumptions \( \mathcal{\mathcal{A}} \). Let \( E \) be an augmented event structure whose \( s \) includes values of the appropriate type for each sensor and free variable appearing in \( e \).

Let \( \Theta^e \equiv \mathcal{\mathcal{E}} \rightarrow \Omega \) (where \( \Omega \) is the set of all value-trees) and \( \Phi^e_\Theta \equiv \mathcal{\mathcal{E}} \rightarrow \mathcal{\mathcal{V}}_T \) be defined by induction on \( e \) in \( E \), so that \( d(e); s(e); \Theta ; \epsilon \vdash e \Downarrow \Phi^e_\Theta(e) \equiv \Theta \mid e \vdash e' \rightarrow e \). Then we say that \( \Phi^e_\Theta : E \rightarrow \mathcal{\mathcal{V}}_T \) is the evaluation of expression \( e \) on \( E \).

Notice that this semantics models round computations \( e \) that are performed with respect to a given \( \Theta \), that cannot change during the local execution. Furthermore, notice that a program may not have any interpretation in an event structure \( \mathcal{\mathcal{E}} \) (i.e., the definition may fail) in case the program was non-terminating in some \( e \) in \( E \). Indeed, a

3 Realising this in practice may require to implement a buffer mechanism on incoming messages, which we abstract away by only focusing on which messages are available in each event.
practical execution of a non-terminating program can never result in an event structure with non-terminating events: if a round of execution starts and does not terminate in a device, from an external point of view the behaviour of that device is indistinguishable from a device that was removed from the network. A non-terminating execution can not be considered an event, as it does not result in sending messages to other devices: thus, our model correctly reflects that those event structures are impossible for such programs.

The meaning of a program $e_{bas}$ can thus be understood according to two intimately related points of view, corresponding to the global-vs-local perspective, called the micro and macro perspective in the context of self-organising systems (Beal et al., 2013; Newton and Welsh, 2004; Gummadi et al., 2005). In the local viewpoint, given by the device-level semantics, an expression $e_{bas}$ is a procedure that, when executed in a round $e$, takes as input the current device identifier $\delta$, the sensor status $\sigma$, and the environment $\Theta$ grouping the message-collections that $\delta$ received from the suppliers of event $e$, and outputs a value $v \in \mathcal{V}$ (the result of $e_{bas}$) and the message-collection $\theta$ to send to neighbours in return.

In the global viewpoint, given by the network-level semantics, the program $e_{bas}$ describes a partial function which, when applied to an augmented event structure $E = (\mathcal{E}, \rightarrow, d, s)$ returns a space–time value $\Phi$ encoding the values produced by the program across space–time.

---

5. Properties

In this section, building on the FXC calculus presented in Section 4, we provide a formal account of the properties of $XC$, and we hint at how further more advanced properties could be conveniently proved in future works.

5.1. Type soundness and determinism

In order to formally state type soundness, program typing needs to be extended to value-tree typing (Fig. 10 top) and configuration typing (Fig. 10 bottom). The former, formalised by judgement $A \vdash e : T$, ensures that both $A \vdash e : T$ holds, and $\theta$ has a structure that could possibly be produced by an evaluation of $e$. The latter, formalised by judgement $A \vdash (\delta, \sigma, \Theta, e, e', \theta) : T$, ensures that every value-tree in $\Phi$ could be an outcome of $e$, and either $\theta$ is empty and $e = e'$ (initial configuration) or $e', \theta$ could be an outcome of $e$ (final configuration).

Final configurations are defined in order to match the shape of a big-step judgement, and thus correspond to an evaluated, terminating program. In the following, we also use the special final configuration $\epsilon$ to model non-terminating executions. Initial configurations represent those with $\epsilon$ also final configurations are those with $\epsilon'$ that are impossible for such programs.

---

The function is partial since a program has no interpretation in an $E$ with events $e$ in which $\Phi$ diverges.

---

4 The function is partial since a program has no interpretation in an $E$ with events $e$ in which $\Phi$ diverges.
the evaluation of a value (which results in itself). We write $c ⊥ r$ to mean that the final configuration $r$ is a possible evaluation result of configuration $c$.

The proofs of the following theorems and lemma rely on the fact that the type system (Fig. 10) enjoys the standard inversion of the typing relation and canonical form properties (cf. Sect. 9.3 of Pierce, 2002).

**Theorem 1 (Type Preservation).** Let $c = (δ, σ, Θ, e, (e, ()))$ be such that $A ⊢ c : T$ holds, $σ$ includes values for every sensor and variable as per assumptions $A$, and assume that $δ; σ; Θ ⊢ e ⊥; θ$ holds for some $w, θ$. Then $A ⊢ w : T$ and $A, e, θ : T$ hold.

**Proof.** We prove the result by induction on the height of tree $Θ$. For expressions $e$ composed of multiple sub-expressions $e_1, ..., e_n$, we define sub-configurations $c_i = (δ, σ, x_i(Θ), e_i, (e_i, ()))$ for $i ≤ n$, together with their evaluation results $w_i, θ_i$ (where applicable).

- **e = x or v or f**: By either rule [E-VAR], [E-VAL] or [E-LIT], $e$ evaluates to $w, ()$ where $v$ is respectively $σ(e)$, $e$ itself, or $e[i]$. All of them trivially have type $T$ by hypothesis on $σ$ and rule [T-XWL]. Notice that this case includes $e = \text{fun } x \Xi[e_1]$, since by hypothesis on $σ$ every free variable in $e$ has to be in $σ$.

- **e = val x = e_1; e_2**: By rules [E-VAL], [T-CONF] and inductive hypothesis, we have that the sub-configurations $c_1$ and $c_2$ have the correct types, thus by rule [T-VAL] and [T-CONF] do $c$ does.

- **e = 0_0(Γ)**: By rule [E-APP], [T-APP] and [T-CONF], $c$ has type $T_i$ for $i ≤ n$. By rule [T-CONF], $Θ$ must consist of value-trees that can be produced by the evaluation of $e$. This ensures that the root of each of those value-trees there is a functional value $t[1]$, and the corresponding sub-tree must be coherent with the application $t(Σ)$. By rule [E-APP], the evaluation of $t(Σ)$ is carried out with respect to $σ_{e_{i+2}}(Θ[1])$, which only contains trees that are compatible with $t[1]$. Thus, we can apply the inductive hypothesis to obtain that $σ_{e_{i+2}} = (δ, σ, σ_{e_{i+2}}(Θ[1]), Ξ(Σ), (i))$ has type $T$ of type $T$. The thesis follows by rule [T-APP] and [T-CONF].

- **e = fun x∴(e_1) (Γ)**: By rule [A-FUN], $e$ evaluates to $e_1[x := \text{fun } x∴(e_1)]$ and the thesis follows by rule [A-FUN] and [T-CONF].

- **e = b(Γ)**: If $b$ is not exchange, the thesis follows trivially by rule [A-BP] and either [A-SENS], [A-UID], [A-SELF] or [A-FOLD]. Assume now that $e = \text{exchange}(u_1, u_2)$. Following rule [A-XC], we construct $v$ by picking values from $Θ$, which must have the correct type, by rule [T-CONF]. The overall result $u, θ$ is then obtained as the result of application $v(c_2)$, which produces a result of the correct type by inductive hypothesis. The thesis follows by rule [A-XC] and [T-CONF].

As a corollary, by induction on the messaging relation, the evaluation of an expression $e$ on an augmented event structure $Σ$ only contains value-trees environments that are well-formed for $e$. This result ensures that evaluation preserves typing, but it does not rule out stuck terms, since they cannot be distinguished from diverging ones by the semantics in Fig. 8: in both cases there is no valid proof tree since there is no explicit modelling of errors (needed to identify stuck configurations) and infinite derivations are not allowed (needed to model non-termination). In fact, this limitation event prevents to formulate a soundness statement at all. This issue is known in the literature since Cousot and Cousot, 1992 and has been analysed in detail in Leroy and Grall, 2009. More recently, automated techniques have been developed that can generate a conservative extension of a big-step inference system, using corules (Ancona et al., 2017a; Dagnino, 2019) to model divergence: Fig. 11 presents such an extension, performed according to the procedure in Ancona et al. (2018) and Dagnino (2022).

The extension introduces an additional configuration $ω$ modelling divergence, with corresponding judgements $δ; σ; Θ ⊢ e ⊥ ω$. Every evaluation rule with premises generates multiple rules for divergence propagation from premises to the conclusion; and a single corule is added for divergence introduction. The judgements derived from this inference system with corules are those having a possibly infinite proof tree using rules only, where nodes of the tree have a finite proof using rules and corule [C-DIV]. This ensures that no judgement $δ; σ; Θ ⊢ e ⊥ ω$ can be derived with a finite proof tree, and conversely, only judgements $δ; σ; Θ ⊢ e ⊥ ω$ can be derived with an infinite proof tree (see Ancona et al., 2018 for further details).

**Theorem 2 (Soundness-May).** Let $c = (δ, σ, Θ, e, (e, ()))$ be a well-typed initial configuration in XC. Then there exist a final configuration $r$ (possibly $∞$) such that $c ⊥ r$.

This result can be proved following step-by-step the proof strategy in Ancona et al. (2017b): by Ancona et al. (2017b), Theorem 3.3, a sufficient condition for soundness is given by the following lemma. Notice that due to the presence of the auxiliary big-step evaluation judgement, in order to know which judgement should be applied when evaluating a configuration, two possible strategies could be used: (i) marking each configuration with a Boolean value, indicating whether the main or auxiliary judgement should be applied; (ii) compiling out the auxiliary judgement, by duplicating rule [E-APP] for each auxiliary rule. In the following proof, we assume that one of these means to detect which judgement to use is in place.

**Lemma 3 (Progress).** Let $WT^∞$ be the set of well-typed initial configurations $c$ with no final configuration $r$ $≠$ $∞$ such that $c ⊥ r$. Given any $c = (δ, σ, Θ, e, (e, ()))$ $∈$ $WT^∞$, then $c ⊥ ω$ is the consequence of a rule where, for all premises of shape $c' ⊥ ω$, $c'' ⊥$ $WT^∞$, and all premises of shape $c' ⊥ r$ with $r$ $≠$ $∞$ are derivable.

**Proof.** The proof is by case analysis over $e$. Let $T$ be the type of $c$, and whenever $e$ has sub-expressions $e_1, ..., e_n$ we consider $c_i = (δ, σ, x_i(Θ), e_i, (e_i, ()))$ for $i ≤ n$.

- **e = x or v or f**: empty case since such expressions $e$ always evaluate to a configuration $r$ $≠$ $∞$ by rules [E-VAR], [E-VAL] or [E-LIT].

- **e = val x = e_1; e_2**: By rule [E-VAL], it must be that either $e_1$ does not evaluate to $r$ $≠$ $∞$ or $e_2[x := v_1]$ does not evaluate to $r$ $≠$ $∞$. The thesis follows by either rule [D-VAL1] or [D-VAL2] respectively.

- **e = 0_0(Γ)**: By rule [E-APP], we have that $c_1$ has type $T_i$ and $z$ has type scheme $T_S < T$. We have the following sub-cases:

  - $∃k$ such that $z_k$ $≠$ $∞$ such that $c_k ⊥ r_k$: let $j$ be less with this property, then $c_k ⊥ r_k$ by rule [D-APP1], with derivable premises $c_i ⊥ r_i$ for $i < j$ and premise $c_j ⊥ ω$ such that $c_j ∈ WT^∞$;

  - $c_j ⊥ r_i = (δ, σ, x_i(Θ), u_i, θ_i)$ for all $i ≤ n$, and $z_k[n_i]$ $≠$ $∞$ such that $n_i$ = $n$, $z(Σ, (i))$ $∉$ $r_i$; then $c_j ⊥ ω$ by rule [D-APP2], with derivable premises $c_i ⊥ r_i$ for $i < n$ and premise $c_j ⊥ ω$ such that $c_j ∈ WT^∞$;

  - $c_j ⊥ r_i$ for all $i ≤ n$: empty case since by rule [E-APP] there is $r$ such that $c ⊥ r$.

- **e = fun x∴(e_1) (Γ)**: By rule [A-FUN], it must be the case that $e_1[x := \text{fun } x∴(e_1)]$ does not evaluate to $r$ $≠$ $∞$. Then $c$ is consequence of rule [D-FUN] with premise in $WT^∞$.

- **e = b(Γ)**: By rules [A-SENS], [A-UID] and [A-SELF] we have that $b$ cannot be a sensor or uid, self as such applications always evaluate to a configuration $r$ $≠$ $∞$. If $b$ is nfold and $c$ does not evaluate to an $r$ $≠$ $∞$, there must be a minimal $j ≤ n$ for which the corresponding premise is not derivable, thus in $WT^∞$. The thesis follows by rule [D-FOLD].

Assume now that $e = \text{exchange}(u_1, u_2)$. By rule [T-XC], we have that $c_1$ has type $T$. By rule [A-XC] we have the following sub-cases:

- $z_k$ $≠$ $∞$ such that $c_k ⊥ r_k$: then $c ⊥ ω$ is the consequence of rule [D-XC] with premise $c_1 ⊥ ω$ such that $c_1 ∈ WT^∞$;
Theorem 4 (Round Determinism). Let $c = (δ, σ, r_1, e_1, e_2, ...) \in \mathbb{E}$ be a well-typed initial configuration in XC. Then there exist a unique final configuration $r$ (possibly infinite) such that $c \Downarrow r$.

Proof. Since the evaluation rules are syntax directed and deterministic, there exists either a single finite proof tree for $c$ or $c \Downarrow r$, or a single infinite proof tree for $c$.

As a consequence of determinism, the soundness may result in Theorem 2 (which ensures that there exists a non-stuck computer, either converging or diverging) becomes equivalent to standard soundness (which requires that there are no stuck computations).

5.2. Expressiveness

In this section, we discuss other formal properties of XC that derive from the ability of XC to express field calculus programs, thus directly inheriting its properties. What is more, XC shares the compositional nature of FC, thus hinting that similar proofs may lead to stronger results, when applied directly to XC programs rather than FC programs.

A recent survey (Viroli et al., 2019) points out properties for subsets of the field calculus (FC) language, including eventual recovery and stabilisation after transient changes (self-stabilisation) (Viroli et al., 2018), independence of the results from the density of devices (Beal et al., 2017), real-time error guarantees (Audrito et al., 2018b), and the ability to express all physically consistent computations (space–time universality) (Audrito et al., 2018a), and to monitor spatio-temporal logic formulas (Audrito et al., 2021b, 2022c). By showing that every FC program can be encoded within XC, we automatically import all these results from literature, allowing for future extensions of them to XC programs not expressible in FC. We conclude showcasing this automatic import process for the paradigmatic case of self-stabilisation.

FC (Viroli et al., 2018) features two separate kinds of values (and types): local values (of local type) and neighbouring values (of field type). A local value models a value $e$ that is not dependent from neighbours, while a neighbouring value represents a neighbour-dependent value, as a map $\phi = \overrightarrow{δ} \mapsto \overrightarrow{v}(\delta)$.

**Definition 4** (Computational Field). A network graph $G = (Δ, \rightarrow, Σ)$ is a finite set of devices $Δ$ with a reflexive neighbouring relation $\leftrightarrow \subseteq Δ \times Δ$ and a sensors-map $Σ : Δ \rightarrow S$. A computational field in $G$ and $V_F$ is a map $Ψ : Δ \rightarrow V_F$ where $V_F$ is the domain of values of type $T$.

Intuitively, $δ_1 \rightarrow δ_2$ (in a given instant of time) represents the possibility for a device $δ_1$ to successfully send a message to another device $δ_2$, thus creating a corresponding messaging $e_1 \rightarrow e_2$ for events $e_1$.
on devices $\delta_i$ ($1 \leq i \leq 2$). Note $\rightarrow$ may be asymmetric—e.g., high-power devices could send messages to far devices with not enough power to reply.

We say that an expression $e$ is self-stabilising iff given a network graph $G$, for any $E$ which eventually conforms to $G$, the result of the program eventually becomes a unique computational field $\psi = \{e\}^{self-stab}(G)$ depending only on $G$, and independent of the specific $E$ considered. Thus, a self-stabilising program can always adapt to a new input (the eventual $G$) given enough time, regardless of its previous history (the specific $E$ before conforming to $G$).

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

**Definition 5 (Stabilising Augmented Event Structure and Limit).** Let $E = (E, \rightarrow, d, s)$ be a countably infinite augmented event structure. We say that $E$ is stabilising to its limit $G = (\Delta, \rightarrow, \Sigma) = \lim E$ iff $\Delta = \{e \in E \mid d(e) = \delta\} \mid e \in E \mid d(e) = \Sigma\{d(c)\}$ and the devices of suppliers are the neighbours of the device of $e$: $\{d(c') \mid d' \rightarrow d\} = \{\delta' \mid d' \rightarrow d(e)\}$.

Although the notion of limit of an event structure provided by Definition 5 is not identical to the notion of limit in analysis; they are intimately connected, justifying the usage of the same name. Notice that although a limit of an event structure $E$ may not exist (thus $E$ not being stabilising), if it does it is unique, as the definition above provides a constructive procedure for $\Delta, \Sigma$ and $\rightarrow$ in terms of what appears infinitely often in $E$.

In concrete deployments, the augmented event structure representing a distributed computation performed over time and across space arises from a network graph (evolving over time) which represents the possible connections across devices in every instant of time. Then, the limit of the event structure is (intuitively) the network graph that is obtained for the time that goes to infinity. A similar notion of limit (and of the event structure) is (intuitively) the network graph that is possible connections across devices in every instant of time. Then, the limit of an event structure $G$ is the set of devices appearing infinitely often in $E$:

$$E = \{\text{nbr}(x, \text{self}(x)) < \text{self}(x)\}.$$  

**Function nbrlt** is defined as **def** nbrlt($x$) $\{\text{nbr}(x, \text{self}(x)) < \text{self}(x)\}.

- any expression not containing an exchange construct, comprising built-in functions (which are self-stabilising by being stateless, not keeping memory or exchanging messages);
- four special forms of the exchange construct, called $nbr$, converging, acyclic and minimising pattern (respectively), defined by restricting both the syntax and the semantic of relevant functional parameters.

We recall here a brief description of the patterns: for a more detailed presentation, the interested reader may refer to Viroli et al. (2018). The semantic restrictions on functions are the following.

**Converging (C)** A function $\tau(x, y, \tau)$ is C with respect to a metric $\text{dist}(\ell_1, \ell_2)$ iff its return value $\tau_y$ has lower maximum distance from $x$ than $\tau_y$: $\text{max}_{\tau}(\text{dist}(\psi(\delta), \psi(\delta))) < \text{max}_{\tau}(\text{dist}(\psi(\delta), \psi(\delta)))$.

**Monotonic non-decreasing (M)** A stateless function $\tau(x, y)$ is M iff it applies pointwise on nvalues and whenever $\tau_1 \leq \tau_2$, also $\tau(\ell_1, \tau) \leq \tau(\ell_2, \tau)$.

**Progressive (P)** A stateless function $\tau(x, y)$ is P iff it applies pointwise on nvalues and $\tau(\ell_1, \tau) \leq \tau(\ell_2, \tau)$.

**Raising (R)** A function $\tau(x, y, \tau)$ is R with respect to total partial orders $<, \leq$ iff: (i) $\tau(\ell_1, \tau) \leq \tau(\ell_1, \tau)$; (ii) $\tau(\ell_1, \ell_2, \tau) \leq \text{min}(\ell_1, \ell_2)$; (iii) either $\tau(\ell_1, \ell_2, \tau) = \ell_1$ or $\tau(\ell_1, \ell_2, \tau) = \ell_2$.

Hence, the four patterns can be described as follows.

**Neighbourhood** This pattern simply corresponds to the nbr function defined in Section 2.2, which accesses values shared by direct neighbours. This pattern self-stabilises, since after stabilisation of its arguments, waiting one more round of executions of every device, the values shared by direct neighbours are stable and so is the result of nbr.

**Converging** In this pattern, variable $x$ is repeatedly updated through function $\tau(c)$ and a self-stabilising value $es$. The function $\tau(c)$ may also have additional (not necessarily self-stabilising) inputs $\tau$. If the range of the metric granting convergence of $\tau(c)$ is a well-founded set of real numbers, the pattern self-stabilises since it gradually approaches the value given by its second argument $es$, reducing the maximum distance appearing in the network at every round of executions of every device.

**Acyclic** In this pattern, the network's values for $x$ are first filtered through a self-stabilising partially ordered "ranking", keeping only values held in devices with lower ranking (thus in particular discarding the device's own value of $x$). This is

5 A function $\tau(\tau)$ is stateless iff its outputs depend only on its inputs and not on other external factors.

6 An ordered set is well-founded iff it does not contain any infinite descending chain.
accomplished by function \texttt{nbrlt}, which returns an nvalue of Booleans selecting the neighbours with lower argument values.

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

**Minimising** In this pattern, the neighbourhood’s values for \( x \) are first increased by a monotonic progressive function \( t^\text{MP} \) (possibly depending also on other self-stabilising inputs). As specified above, \( t^\text{MP} \) needs to operate pointwise on nvalues.

Afterwards, the minimum among those values and a local self-stabilising value is then selected by function \texttt{nfold}. Finally, this minimum is fed to the \texttt{raising} function \( r^\text{k} \) together with the value of \texttt{self(x)} (and possibly any other inputs \( \mathfrak{b} \)), obtaining a result that is higher than at least one of the two parameters. We assume that the partial orders \( <,\leq \) are Noetherian,\(^7\) so that the raising function is required to eventually conform to the given minimum.

Intuitively, this pattern self-stabilises since it computes the minimum among the local values \( \ell = \text{self}(es) \) passed as third argument of \texttt{nfold}, after being increased by \( t^\text{MP} \) along every possible path (and the effect of the raising function can be proved to not affect the final result after stabilisation).

**Theorem 5.** Expressions \( es \) as in Fig. 12 are self-stabilising.

**Proof.** Let \( \Xi = (E,\rightarrow,\prec,\delta) \) be any augmented event structure with a given limit \( G \). The proof proceeds by induction on the syntax of expressions \( es \), which could be:

- A variable, \texttt{fun} -expression, exchange-free local literal or nvalue: in all those cases, the expression is constant hence self-stabilises as soon as the event structure starts to conform to the limit graph.
- A \texttt{val} -expression \texttt{val x = es1;es2}. By inductive hypothesis, the sub-expression \( es1 \) stabilises to \( \Psi \) after a certain event. Then, \texttt{val x = es1;es2} evaluates to the same value as the expression \( es1 \) after adding \( \Psi \) to \( \Sigma \), which is self-stabilising by inductive hypothesis. Thus, the whole \texttt{val} -expression self-stabilises.
- A functional application \texttt{es(val(\mathfrak{b}))}. By inductive hypothesis, all sub-expressions self-stabilise to \( \Psi,\Xi \) after a certain event. First, assume that \( \Psi(\delta)(\delta) = 1 \) is a built-in (which cannot be an \texttt{exchange} by the posed syntax restrictions) or sensor. Then \( t(\Xi) \) stabilises immediately as built-ins and sensors do not depend on the environment \( \Theta \). Otherwise, \( t \) has to be a \texttt{fun} -expression \texttt{fun x(\mathfrak{a}) \{ es' \}} and thus \( t(\Xi) \) evaluates to the same value as \( es' \) after adding \( x \rightarrow \Psi,\Xi \rightarrow \Psi,\Xi \) to \( \Sigma \), which is self-stabilising by inductive hypothesis.
- A neighbourhood pattern \texttt{nbr(es1,es2)}. By inductive hypothesis, expressions \( es1,es2 \) self-stabilise to \( \Psi,\Psi \) after a certain event. Then, \texttt{nbr(es1,es2)} self-stabilise to the corresponding nvalue after one additional full round of execution of each device.
- A converging pattern \texttt{es} = \texttt{exchange(e(x),A) \Rightarrow resend }f_{Cy(x,es,\mathfrak{b})}. By inductive hypothesis, \( es \) self-stabilises to \( \Psi \) after a certain event \( c_0 \). Given any index \( n \), let \( d_c \) be the maximum distance between \( es(\delta') \) and \( \Psi(\delta)(\delta) \) for all devices \( \delta' \) and events \( \epsilon \) with \( \delta = d_\epsilon \) of the \( n \)th round of execution of every device after \( c_0 \). We prove that \( d_c \) is strictly decreasing with \( n \). Since distances are computed on a well-founded set, it follows that they have to become zero for a sufficiently large \( n \), proving self-stabilisation of \( es \) to \( \Psi \).

Consider an event on the \( n \)th round of execution of every device after \( c_0 \). Thus, supplier events belong to rounds of execution \( \geq n + 1 \), hence their distance with \( \Psi \) is at most \( d_{n+1} \). By definition of converging function, it follows that \( es \) is strictly closer to \( \Psi \) than \( d_{n+1} \), concluding the proof.

- An acyclic pattern \( es = \texttt{exchange(e(x),A) \Rightarrow resend }f_{Cy(x,es,\mathfrak{b})}. \) By inductive hypothesis, \( es,es,es,\mathfrak{b} \) self-stabilise to \( \Psi,\Psi,\Psi,\Xi \) after a certain event \( c_0 \). Let \( e \) be any event after \( c_0 \) of the device \( d_0 \) of minimal ranking \( \Psi(d_0)(\delta_0) \) in the network. Since \( \Psi(d_0)(\delta_0) \) is minimal, the \texttt{nbrlt(es)} is false, the \texttt{mux}-expression reduces to \( es \) and the whole \( es \) to \( es(es,\mathfrak{b}) \), which self-stabilises by inductive hypothesis after \( c_1 \). Similarly, in any event of the second-minimal ranked device after \( c_1 \) the \texttt{mux}-expression is stable, thus ensuring stabilisation after a certain event \( c_2 \). The thesis follows by repeating the same reasoning on all devices in order of increasing ranking.

- A minimising pattern \( es_m = \texttt{exchange(e(x),A) \Rightarrow resend }f_{Cy(x,es,\mathfrak{b})}. \) By inductive hypothesis, \( es,\mathfrak{b} \) self-stabilise to \( \Psi,\Xi \) after \( c_0 \). Let \( P = \mathfrak{b} \) be a path in the network graph \( G \), and define its weight as the result of picking the eventual value \( \ell = \Psi(\delta) \) of \( es \) in the first device \( \delta_1 \), and repeatedly passing it to subsequent devices through the monotonic progressive function, so that \( \ell_{+1} = t^\text{MP}(\ell,\Psi(\delta_1)) \). Notice that the weight is well-defined since function \( t^\text{MP} \) is required to be stateless. Finally, let \( \Psi_{\text{out}}(\ell) \) be such that \( \Psi_{\text{out}}(\ell) = \ell \) is the minimum weight for a path \( P \) ending in \( \mathfrak{b} \).

Let \( \delta,\delta' \) be the list of all devices \( \delta \) ordered by increasing \( \Psi_{\text{out}}(\ell) \). Notice that the path \( P \) of minimal weight \( \ell_{\delta'} \) for device \( \delta' \) can only pass through nodes such that \( \ell_{\ell} \leq \ell_{\delta'} \) (thus s.t. \( \ell < \delta \)). In fact, whenever a path \( P \) contains a node \( j \) the weight of its prefix until \( j \) is at least \( \ell_{\delta'} \), and any longer prefix has strictly greater weight than \( t^\text{MP} \) is progressive.

We prove by complete induction on \( i \) that after a certain event \( \ell_{i+1} \), expression \( es_m \) stabilises to \( \Psi_{\text{out}}(\ell') \) in device \( \delta' \), and assumes values \( \geq \Psi_{\text{out}}(\ell') \) in devices \( \delta' \) with \( \ell > \delta' \). By inductive hypothesis, assume that devices \( \delta' \) with \( j < i \) are all self-stabilised after \( c_i \). After every device performed an additional round, these values are available in the \( x \) variable of \texttt{exchange}. Consider the evaluation of the expression \( es_m \) in a device \( \delta' \) with \( j \geq i \). Since argument \( es \) of \texttt{nfold} is also the weight of the single-node path \( P = \delta' \), it has to be at least \( \ell \geq \ell_{\delta'} \geq \ell_{j'} \). Similarly, the second argument \( v \) of \texttt{nfold} for devices \( \delta' \) with \( j < i \) has to be at least \( \omega(\ell') \geq \ell_{j'} \geq \ell_{j''} \) since it corresponds to weights of (not necessarily minimal) paths \( P \) ending in \( \ell' \) obtained by extending a minimal path for a device \( \delta' \) with \( j < i \) and the whole \( \Psi_{\text{out}}(\ell') \) in device \( \delta' \) with \( j \geq i \), since \( t^\text{MP} \) is progressive.

Thus, as long as the minimum value for the whole expression among non-stable devices is lower than \( \ell_{\delta'} \), the result of the \texttt{nfold} subexpression is strictly greater than this value. The same holds for the overall value, since it is obtained by combining the output of \texttt{nfold} with the previous value for \( x \) through the rising function \( r^\text{k} \), and a rising function has to be equal to the first argument (the \texttt{nfold} result strictly greater than the minimum), or \( \triangleright \) the second. In the latter case, it also needs to be greater or equal to the first argument (again, strictly greater than the minimum) or strictly greater than the second argument\(^8\) (not below the minimum value).

Thus, after a round of execution of every device, the minimum value among non-stable devices has to increase, until it eventually

\(^7\) A partial order is Noetherian iff it does not contain any infinite ascending chains.

\(^8\) It cannot be equal to the second argument, as it is \( \triangleright \)-greater than it.
surfaces $\ell_\delta$ since $\delta$ is Noetherian. From that point on, that minimum cannot drop below $\ell_\delta$, and the output of \texttt{nfold} in $\delta^i$ stabilises to $\ell_\delta$. In fact, if $P$ is a path of minimum weight for $\delta^i$, then either:
- $P = \delta^i$, so that $\ell_\delta$ is exactly the third argument of \texttt{nfold}, hence also the output of it (since the second argument is greater than $\ell_\delta$).
- $P = Q, \delta^i$ where $Q$ ends in $\delta^j$ with $j < \delta^i$. Since $\ell_\delta$ is monotonic non-decreasing, the weight of $Q', \delta^i$ (where $Q'$ is minimal for $\delta^i$) is not greater than that of $P$; i.e., $P' = Q', \delta^i$ is also a path of minimum weight. It follows that $\nu(\delta^i)$ (where $\nu$ is the second argument of \texttt{nfold}) is $\ell_\delta$.

Since the order $\preceq$ is Noetherian, the rising function on $\delta^i$ has to eventually select its first argument. Thus, it will select the output of the \texttt{nfold} subexpression, which is $\ell_\delta$, and from that point on the minimising expression will have self-stabilisation on device $\delta^i$ to $\ell_\delta$, and every device $\delta^i$ with $j \geq i$ will attain values $\geq \ell_\delta$, concluding the inductive step and the proof. □

6. Implementation

To showcase the broad applicability of the proposed language design, we implemented a Scala and a C++ version of XC. The Scala version has been developed as an extension of ScaFi (Casadei et al., 2022b), and aims at showcasing the DSL and maximise portability to different platforms, including simulators. The C++ version has been developed as an extension of FCPP (Audrito et al., 2020), and has consequently been integrated into the main FCPP distribution. This version targets performance and devices with limited resources (Testa et al., 2022; Audrito et al., 2023b).

6.1. Scala DSL

We provide an implementation of XC as a DSL embedded into the Scala language because of its cross-platform support (Doeraene, 2018), popularity for building distributed systems (Ghosh et al., 2012), and advanced support for internal DSLs (Artho et al., 2015). This implementation has been developed as an extension of ScaFi (Casadei et al., 2022b). The DSL is organised into a few core XC constructs and a library of reusable functions. The core constructs (cf. Fig. 6) are declared by a Scala trait with the following interface:

\begin{verbatim}
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]
}
\end{verbatim}

The if/else of XC is modelled as a \texttt{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.

\begin{verbatim}
class NValue[T](val defaultMessage: T, val customMessage: Map[ID,T] = Map.empty) {
  def fold[V>T](init: V):(V,V) => V: NValue[V] = ...
  def map2[R,S](other: NValue[R]): Message[T,R]=>S = ...
}
\end{verbatim}

The Scala DSL is publicly available under the Apache 2.0 licence at https://github.com/scafi/artifact-2021-ecoop-xc and permanently as an archived artefact on Zenodo (Casadei, 2022b).

An XCProgram[T] models a single local computation. As discussed (Section 2.1), a XC system consists of multiple devices repeatedly (i) acquiring context, (ii) computing the round, and (iii) propagating messages to neighbours. The execution environment provides a context comprising sensor values (e.g., for the built-in sensing functions—cf. Fig. 2) and messages from the neighbours. For instance, the following code shows the execution on a device:

\begin{verbatim}
while(true) {
  val sensorData = getData() // implementation-specific
  val messagesFromNeighbours = getMessages() // implementation-specific
  val context = Context(sensorData, messagesFromNeighbours)
  val process(output, messageCollection) = gradient.fire(context)
  propagate(messageCollection) // implementation-specific
}
\end{verbatim}

In this implementation, message communication occurs only \texttt{before} (Line 3) and \texttt{after} (Line 7) the firing (Line 5) to ensure that the \texttt{exchange} within the round are all executed atomically w.r.t. the messages that are received and sent by the device (Section 2.1). Concrete details of a system implementation depend 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 message passing), or a simulator (where physical and/or logical devices are virtualised) (Casadei et al., 2020). What these implementations must do in order to support a XC system is providing the implementation-specific functions of the listing above: \texttt{getData()} to obtain values from the local environment, \texttt{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), \texttt{process()} to drive actuations (e.g., locally on a node, or through a command on a cloud back-end), and \texttt{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).
6.2. C++ DSL

We implemented XC as a C++ DSL,\(^\text{10}\) by extending FCPP (Audrito, 2020). This implementation is designed for \(i\) efficiency, and \(ii\) custom architectures. For \(i\), we rely on C++’s compile-time optimisation and execution on the bare metal. We also performed careful profiling to manually optimise 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 Fig. 13 for a code sample). This implementation has been already used and tested on several different back-ends, including:

- batch simulations of distributed networks, producing plots sum-marising the network behaviour across several runs (Audrito, 2020);
- graphical and interactive 3D simulation of a distributed network through OpenGL (Audrito et al., 2022e);
- processing of XC algorithms on large graph-based data in HPC (Audrito et al., 2022d);
- deployment on microcontroller architectures with either Contiki OS or MIOSIX (Testa et al., 2022; Audrito et al., 2023b), also interoperable with Android phones (work in progress).

No external dependencies are needed for those back-ends.

7. Evaluation

In this section, we evaluate XC.\(^\text{11}\) 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 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.

\(^{10}\) The C++ DSL is publicly available under the Apache 2.0 licence at: https://fcpp.github.io.

\(^{11}\) The simulation framework, its description, and instructions for reproducing the experiments are publicly available at https://github.com/scali/artifact-2021-ecoo-smartc and permanently as an archived artefact on Zenodo (Casadei, 2022a). This artefact is also described by the artefact paper (Audrito et al., 2022b).
level locally perceived by sensors. Blue squares are nodes in the channel from detector to operations centre.

7.2. Results

Fig. 15 shows that the mean warning received by the operations centre (blue) during a run approximates the actual warning in the inspected area (magenta). The jaggs 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).

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).

7.3. Comparison with other programming models

In order 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.12 These two functions are typical patterns of self-organisation as well as paradigmatic examples for the presented language, for they show, respectively, (i) how data exchange is integrated into computations and (ii) compositionality, namely how the approach enables composition of self-organising behaviours that involve both interaction and processing. The comparison revolves around the analysis of the resulting code of the implementations of each example for the different paradigms, using basic quantitative code metrics (such as Lines of Code (LoC) and number of occurrences of specific language constructs used) and also qualitative arguments (e.g., related to the structure of each implementation).

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 (Viroli et al., 2019), no systematic comparison (e.g. via formal translation) with other approaches has been previously carried out. There are two challenges for a rigorous comparison: (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 (namely, the “execution model”—cf. Section 4.3), which are brittle to neatly separate in other approaches. To address these challenges, we have considered multiple design, focussing on a comparison involving (i) a pub-sub “idiomatic” solution, $SP_{PS}$, (ii) a pub-sub XC-like solution $SP_{PSXC}$ with a design inspired by XC, and (iii) an XC solution $SX_{XC}$. 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 $SP_{PS}$, the logic spreads over multiple subscription handlers, while in $SP_{PSXC}$ and $SX_{XC}$ the core logic is neatly separated. The $SP_{PS}$ version uses 4 handlers (and crucially, any additional field would need a further handler), 2 $\&$ands, and 4 $\&$ublishes, while $SP_{PSXC}$ uses 2, 1, and 3 resp. Also, $SP_{PS}$ keeps 6 state variables for the input context of a device--$SP_{PSXC}$ only 3. W.r.t. $SX_{XC}$, $SP_{PSXC}$ 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 $SP_{PS}$ 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.

8. Related work

XC can be framed in the context of a long-term research thread on programming languages and tools for programming collective adaptive systems, known under the umbrella terms of field-based coordination (Mamei and Zambonelli, 2006; Viroli et al., 2019) and aggregate computing (Beal et al., 2015; Viroli et al., 2019). This research area is characterised by works on formal calculi (Audrito et al., 2019, 2023a), new constructs (Audrito et al., 2020; Casadei et al., 2019), and libraries for the implementation of programs and computations (Viroli et al., 2018; Beal et al., 2017; Audrito et al., 2018a), and libraries for the implementation of tools for distributed monitoring (Pianini et al., 2013; Audrito et al., 2022). As discussed in Section 5.2, by subsuming other field constructs, reusing existing simulators (Aguzzi et al., 2013; Audrito et al., 2022), algorithms and patterns (Beal, 2009; Audrito et al., 2017; Pianini et al., 2021b; Audrito et al., 2021a; Pianini et al., 2022), execution models (Pianini et al., 2021a), distributed platforms and deployments (Casadei et al., 2020, 2022b), and libraries for application domains such as swarm robotics (Aguzzi et al., 2023) and distributed monitoring (Audrito et al., 2021b).

In a nutshell, this work proposes a new calculus, XC, inspired by previous calculi, that subsumes them and is strictly more expressive in its ability to model messages differentiated on a neighbour basis (see Section 8.3 for a more detailed comparison). The implementations of XC covered in Section 6 are integrated into the existing Scala/C++ toolchains, hence providing the ability to reuse existing simulators (cf. the use of Alchemist Pianini et al., 2013 in the case study artefact Casadei, 2022a). As discussed in Section 5.2, by subsuming other field constructs, reusing existing libraries (Aguzzi et al., 2023; Casadei et al., 2021; Audrito et al., 2021b) is also straightforward.

We organise related work by first providing a high-level perspective on field-based coordination (Section 8.1). Next we describe approaches based on ensembles and attribute-based communication (Section 8.2), which are close to our solution but adopt fundamentally different design choices. Finally, we compare in detail with field calculi (Section 8.3) and briefly discuss abstraction and compositionality (Section 8.4).
8.1. Field-based coordination

Field-based coordination, as a paradigm to develop self-organising systems, originate from two main research areas: spatial computing (DeHon et al., 2007), where the idea of aggregate computing (Beal et al., 2015) emerged, and coordination models and languages (Malone and Crowston, 1994). Two surveys cover these two perspectives. The work in Beal et al. (2013) 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, Virolı et al. (2019) 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 Virolı et al. (2018) and Audrito et al. (2020) 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.

8.2. Ensembles and attribute-based communication

Recently, field-based coordination is also framed as a paradigm for collective adaptive systems (CASs) (Ferscha, 2015), which is a further application target for self-organisation techniques in general. There, related approaches include ensemble-based engineering (Bures et al., 2013; De Nicola et al., 2014) and attribute-based communication (Abd Alrahman et al., 2020). 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 (De Nicola et al., 2014), 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 populated groups of target components; actions provide ways to read, retrieve, put information, and to create new components. AbC (Abd Alrahman et al., 2020) 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.

8.3. Field calculi

Field calculi, surveyed in Virolı et al. (2019), 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) (Virolı et al., 2018; Audrito et al., 2020), which is the reference model for computational fields (Virolı et al., 2019), also implemented by DSLS like ScaFi (Casadei et al., 2022b, 2021) and FCPP (Audrito, 2020; Audrito et al., 2022e). 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 $\nu = \langle \bar{v} \rightarrow \hat{v} \rangle$. In particular, local values are equivalent to nvalues $\nu$ 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 ($\text{nbr}$, $\text{rep}$ Virolı et al., 2018 and $\text{share}$ Audrito et al., 2020) can be modelled within XC: $\text{nbr}$ is the same defined function introduced in Section 2.2, just restricted to operate on local values only; $\text{rep}$ corresponds to an exchange with $\text{send}$ restricted to operate on local values only; and $\text{rep}(\nu, (x) \Rightarrow (\bar{v} := \text{self}(\nu)))$ can be translated to $\text{exchange}(\nu, (x) \Rightarrow (\bar{v} := \text{self}(\nu)))$. Notice that the converse translation is not possible, as $\text{nbr}$, $\text{rep}$ or $\text{share}$ expressions with arguments of neighbouring type have no defined behaviour in FC. Thus, $\text{nbr}$ and $\text{exchange}$ in XC are strictly more expressive than their corresponding FC counterparts $\text{nbr}$ and $\text{share}$: 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 Virolı et al. (2019), include eventual recovery and stabilisation after transient changes (self-stabilisation) (Virolı et al., 2018), independence of the results from the density of devices (Beal et al., 2017), real-time error guarantees (Audrito et al., 2018b), efficient monitorability of spatio-temporal logic properties (Audrito et al., 2021b, 2022c), and ability to express all physically consistent computations (space–time universality) (Audrito et al., 2018a). 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 — as done for self-stabilisation in Section 5.3.

The neighbours calculus (Audrito et al., 2023a) is a variant of FC that aims to simplify embeddability of DSLS inside general-purpose host languages by not requiring to deal with fields at the type level. This is achieved by considering a primitive folding operator inside which $\text{nbr}$ expressions can be evaluated.

8.4. Shared memory models

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 Beal et al., 2013; Virolı et al., 2019) or shared memory models (surveyed, e.g., in Morin and Puaut, 1997; Mushtaq et al., 2011).

By a modelling and implementation perspective, an XC system could be thought of as consisting of devices where each device has its own portion of shared memory, e.g., as a tuple space (refer to Virolı et al., 2019 for a historical account of the evolution from tuple-based to field-based coordination). Then, interaction in XC could be realised in at least three main ways: (i) by writing and reading data from a global tuple space, (ii) by distributing the tuple space into device-specific tuple spaces, then writing data on neighbours’ tuple spaces, and reading data from the local tuple space, or dually, (iii) by writing data on the local tuple space, and reading data from neighbours’ tuple spaces. One important difference, however, is that whereas read and write operations in shared memory models usually have an explicit target location (or, in centralised solutions, explicit parameters in tuple templates for retrieval via pattern matching), in XC these targets are implicit and depend on the program’s runtime structure (cf. the notion of alignment discussed in Section 2.4). Among tuple-based models and systems, LIME (Linda in Mobile Environments) (Murphy et al., 2006) seems especially related (also in motivation and target systems), with its idea of replacing a global tuple space with “transiently shared tuple spaces” whose contents are automatically populated based on connectivity (cf. neighbouring relationship in XC). A detailed investigation of similar approaches, left as a future work, can be especially instrumental for addressing the problem of building a middleware for XC systems.
9. Conclusion and outlook

In this article, we present and discuss the design of XC, a programming language for homogeneous distributed systems that abstracts over common issues in developing distributed applications, including faults, loss of messages, and asynchronicity. The design of XC is minimal, featuring only one communication primitive. Despite its simplicity, we show that XC can capture several communication patterns for self-organising and distributed collective systems and it is effective for writing software controlling large-scale collections of devices.

The design of XC, through nvalues and the new semantic construct exchange, promotes interesting directions for future work. A first direction is to develop comprehensive libraries of reusable XC blocks: works such as Viroli et al. (2018) define combinators, i.e., general field functions implementing key behavioural elements of information diffusion, collection, and degradation—the composition of which enables the definition of a wide spectrum of higher-level and application-specific functions. We plan to devise new such building blocks with XC, e.g., to realise sparse choice of leaders (Pianini et al., 2022), consensus (Beal, 2016), and concurrent collective processes (Casadei et al., 2021). Secondly, we are currently assessing the impact of XC constructs on real-world application programming, thanks to our porting in Scala and C++.

CRediT authorship contribution statement

Giorgio Audrito: Conceptualisation, Methodology, Investigation, Software, Data curation, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. Roberto Casadei: Conceptualisation, Methodology, Investigation, Software, Data curation, Validation, Visualization, Writing – original draft, Writing – review & editing. Ferruccio Damiani: Conceptualisation, Methodology, Supervision, Writing – review & editing, Funding acquisition. Guido Salvaneschi: Conceptualisation, Supervision, Writing – review & editing, Funding acquisition. Mirko Viroli: Conceptualisation, Methodology, Supervision, Writing – review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The link to relevant data artifacts can be found in the paper.

Acknowledgements

This publication is part of the project NODES which has received funding from the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036. This study was carried out within the Agritech National Research Center and received funding from the European Union Next-GenerationEU of PNRR, MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1032 17/06/2022, CN00000022. The work was also partially supported by the Italian PRIN project “CommonWears” (2020HCWLP), by the Ateneo/CSP ex-post 2020 project NewEdge, and by the Swiss National Science Foundation (SNSF), grant 200429. This manuscript reflects only the authors’ views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

Best paper at COORDINATION’17.
Giorgio Audrito is an assistant professor at University of Turin (Italy). His research interests include distributed computing, programming languages, distributed algorithms and graph algorithms; as well as innovative didactic methods through gamification. Since 2013 he is the team leader of the Italian team at the International Olympiad in Informatics. Since 2020 he is the original designer, main developer and maintainer of FCPP, a C++ framework for aggregate programming, which is winner of a best artefact and an outstanding artefact awards.

Roberto Casadei is an assistant professor at Alma Mater Studiorum–Università di Bologna (Italy). His research revolves around software engineering and distributed artificial intelligence. He has 60+ publications in international journals and conferences on topics including collective intelligence, aggregate computing, self-* systems, and IoT/CPS. He also leads the development of the open-source ScaFi aggregate programming toolkit. He has been serving in the OC/PC of multiple conferences such as ACSOS, COORDINATION, ICCCI, and SAC, and as editorial board member of JAISCR.

Guido Salvaneschi has been an Associate Professor at the University of St.Gallen (Switzerland) since September 2020. He earned his Ph.D. from the Dipartimento di Elettronica e Informazione at Politecnico di Milano, under the supervision of Prof. Carlo Ghezzi. In 2011, he was a Visiting Ph.D. student at MIT’s Computer Science and AI Laboratory, under the supervision of Prof. Barbara Liskov. He served as an Assistant Professor at TU Darmstadt until Fall 2020. Guido’s research interests include programming languages and software engineering techniques for distributed systems.

Mirko Viroli is Full Professor in Computer Engineering at Alma Mater Studiorum–Università di Bologna (Italy). He is an expert in advanced software development and engineering, author of more than 250 papers (of which more than 70 on international journals). He is member of the Editorial Board of IEEE Software magazine.

Ferruccio Damiani is an full professor at the Computer Science Department of the University of Turin (Italy). There, he founded and coordinates the System Modelling, Verification and Reuse (MoVeRe) research group, whose goal is to contribute to an effective seamless integration of Formal Methods into software and system development methodologies. He is a member of the board of the European Association for Programming Languages and Systems (EAPLS).