The Journal of Systems and Software 210 (2024) 111976

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

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 targetting.

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 Welsh, 2004; Sene Júnior et al., 2022), that notably include *spatial computing* (DeHon 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

☆ Editor: Dr. Nicole Novielli.

* Corresponding author.
E-mail addresses: giorgio.audrito@unito.it (G. Audrito), roby.casadei@unibo.it (R. Casadei), ferruccio.damiani@unito.it (F. Damiani), guido.salvaneschi@unisg.ch (G. Salvaneschi), mirko.viroli@unibo.it (M. Viroli).

https://doi.org/10.1016/j.jss.2024.111976
Received 9 March 2023; Received in revised form 10 November 2023; Accepted 19 January 2024
Available online 20 January 2024
0164-1212/© 2024 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

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 actuations, 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.

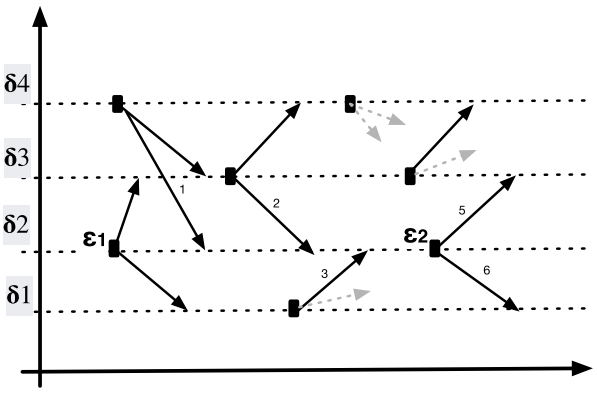


Fig. 1. XC system model.

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 δ_1 to δ_4. In the considered time span, device δ_2 wakes up twice and performs two computation rounds, ϵ_1 and ϵ_2. Grey arrows denote messages that get lost and are never received. The computation ϵ_2 processes three messages, received from δ_4, δ_3, and δ_1 while δ_2 was asleep. After the computation, δ_2 sends out a message to δ_3 and to δ_1. The order of messages from a same sender is preserved but, other than that, there are *very few assumptions on messages*. If a device δ_1 runs multiple rounds before a device δ_2 even runs a single round, δ_2 sees only the message received from the last round of δ_1, i.e., newly received messages from a same sender overwrite older ones. Also, messages are not removed from the buffer after reading them, unless they *expire* (i.e., are deemed too old according to any pre-established criterion) or unless they are replaced by a new message from the same device, allowing messages to (possibly) persist across rounds. The XC design abstracts over the specific expiration criteria: common choices include removing messages after each read, or after a validity time elapses. This time interval is highly application-specific and stems from a trade-off

[1] This approach is often referred to as *macroprogramming* (Casadei, 2023; Newton and Welsh, 2004) or *multi-tier programming* (Weisenburger 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 δ 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 δ; *(ii)* messages may get lost; *(iii)* devices may disappear or fail; *(iv)* a device may reboot, losing its queue of received messages; *(v)* δ may deem messages from some devices to be expired. Crucially, XC does not require distinguishing among those cases. When a device wakes up, it finds some messages from (the most recent available execution round of) some other devices. The devices for which a message is available in a certain round are *the neighbours* for that round.

This system model and the terminology associated to it (e.g., 'send message to a neighbour') is adopted throughout the paper. These design choices make XC agnostic to the actual communication channel, topology creation and discovery mechanism: e.g., push or pull, broadcast or point-to-point. For example, the same programming model would apply even if a device, after waking up, contacts the neighbours to fetch their current value in a *pull* fashion. Instead, in a network of micro-controller devices, Bluetooth 5.0 *extended advertisements* could be used to share data with neighbour devices in physical proximity, without an explicit discovery mechanism, as the topology is induced by the messages that are actually received. Such an implementation would also grant *causal consistency* (Ahamad 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 ℓ include traditional types A like float, string or list. Neighbouring values (*nvalues*) are a map $\underline{w}$ from device identifiers δ_i to corresponding local values ℓ_i, with a *default* ℓ, written $\ell[\delta_1 \mapsto \ell_1, \ldots, \delta_n \mapsto \ell_n]$. A nvalue is used to describe the (set of) values received from and sent to neighbours. In highly decoupled distributed systems *only a few* neighbours may *occasionally* produce a value. The devices with an associated entry in the nvalue are hence usually a subset of all devices, e.g., because a device is too far to provide a value or the last provided value has expired. The default is used when a value is not available for some reason as will be discussed later (e.g., if a device just appeared and has not yet produced a value). For this reason, it is convenient to adopt the notation above and read it "the nvalue $\underline{w}$ is ℓ everywhere (i.e., for all neighbours) except for devices $\delta_1, \ldots, \delta_n$ with values $\ell_1, \ldots, \ell_n$". To exemplify nvalues, in Fig. 1, upon waking up for computation ϵ_2, δ_2 may process a nvalue $\underline{w} = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2, \delta_1 \mapsto 3]$, corresponding to the messages carrying the scalar values 1, 2, and 3 received when asleep from δ_4, δ_3, and δ_1. The entries for all other devices default to 0. After the computation, δ_2 may send out the messages represented by the nvalue $\underline{w}' = 0[\delta_3 \mapsto 5, \delta_1 \mapsto 6]$; so that 5 is sent to δ_3, 6 is sent to δ_1, and 0 is sent to every other device (such as a newly-connected device with no dedicated value yet). We also use the notation $\underline{w}(\delta')$ for the local value ℓ' if $\delta' \mapsto \ell'$ is in $\underline{w}$, or the default local value ℓ of $\underline{w}$ otherwise, reflecting the interpretation of nvalues as maps with a default. For instance, $\underline{w}'(\delta_1) = 6$ and $\underline{w}'(\delta_2) = 0$. To help the reader, in code snippets, we underline the variables holding neighbouring values, and, similarly, we underline a primitive type $\underline{A}$ to indicate the type of an nvalue $\underline{w} = \ell[\delta_1 \mapsto \ell_1, \ldots, \delta_n \mapsto \ell_n]$ where $\ell, \ell_1, \ldots \ell_n$ have type A.

Nvalues generalise local values. A local value ℓ can be automatically converted to a nvalue $\ell[]$ with a default value for every device. In fact, the distinction between local values and nvalues is only for clarity: local values can be considered equivalent to nvalues where all devices are mapped to a default value. In the 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 $\underline{w}_1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2]$ and $\underline{w}_2 = 1[\delta_4 \mapsto 2]$, we have $\underline{w}_3 = \underline{w}_1 + \underline{w}_2 = 1[\delta_4 \mapsto 3, \delta_3 \mapsto 3]$. Note that $\delta_3 \mapsto 3$ in $\underline{w}_3$ is due to the fact that $\delta_3 \mapsto 2$ in $\underline{w}_1$ and δ_3 has default value 1 in $\underline{w}_2$. Using also the automatic promotion of local values to nvalues, we have that $\underline{w}_1 + 1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2] + 1 = 1[\delta_4 \mapsto 2, \delta_3 \mapsto 3]$.

Operations on nvalues. Besides pointwise manipulation, nvalues can be folded over, similar to a list, through built-in function `nfold`$(f : (A, B) \to A, \underline{w} : \underline{B}, \ell : A) : A$, where the function f is repeatedly applied to neighbours' values in a field $\underline{w}$ (thus excluding the value for the *self* device), starting from a base local value ℓ. For instance, assume that δ_2 is performing a `nfold` operation, with the current set of neighbours $\{\delta_1, \delta_3\}$. Then `nfold`$(+, \underline{w}_1, 10) = 10 + \underline{w}_1(\delta_1) + \underline{w}_1(\delta_3) = 10 + 0 + 2$, where $\underline{w}_1 = 0[\delta_4 \mapsto 1, \delta_3 \mapsto 2]$ is as above. As nvalues should be agnostic to the ordering of the elements (i.e., the ordering of the identifiers δ'), we usually assume that f is associative and commutative.

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

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

```
1 def distanceEstimate(n) { // type: (num) → num
2   nfold(min, n + senseDist, Infinity)
3 }
```

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

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

XC operators on nvalues behave uniformly on neighbours to encourage uniform behaviour on each element of a nvalue. This approach is idiomatic in XC and results in a more resilient behaviour – inherently tolerate changes of neighbourhoods between rounds. Yet, non-uniform behaviour can be encoded via built-in function `uid` (combined with communication primitives, Section 2.3), which provides the unique identifier δ 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 `mux`(ℓ_1, ℓ_2, ℓ_3) returns ℓ_2 if ℓ_1 is `True`, ℓ_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.

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

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

2.3. Communication in XC: Exchange

XC features a single communication primitive exchange(e_i, ($\underline{n}$) => return e_r send e_s) which de-sugars to exchange(e_i, ($\underline{n}$) => (e_r, e_s)) and is evaluated as follows. *(i)* the device computes the local value ℓ_i of e_i (the *initial* value). *(ii)* it substitutes variable $\underline{n}$ with the nvalue $\underline{w}$ of messages received from the neighbours for this exchange, using ℓ_i as default. The exchange returns the (neighbouring or local) value v_r from the evaluation of e_r. *(iii)* e_s evaluates to a nvalue $\underline{w}_s$ consisting of local values to be sent to neighbour devices δ', that will use their corresponding $\underline{w}_s(\delta')$ as soon as they wake up and perform their next execution round.

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

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

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

```
1 def ping-pong() { // type: () → num
2   exchange( 0, (n) => retsend n + 1 )
3 }
```

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

Example 3 (*Self-organising Distance*)**.** Computing the minimum distance from any device 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).

```
1 def distanceTo(src) { // type: (bool) → num
2   exchange( Infinity, (n) => retsend mux(src, 0,
        distanceEstimate(n)) )
3 }
```

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 mux (i.e., a strict version of if that computes both its branches, and then selects the output of one of them as result based on the condition) is needed, as sources, though returning 0, must also evaluate function call distanceEstimate (thus sharing their value $\underline{n}$). Any change in the network (e.g., due to failure, mobility, dynamic joining) directly affects the domain of $\underline{n}$, hence the local computation and eventually the whole network—resulting in inherent adaptiveness.

2.4. Compositionality through alignment

If a program executes multiple exchange-expressions, XC ensures by *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:

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

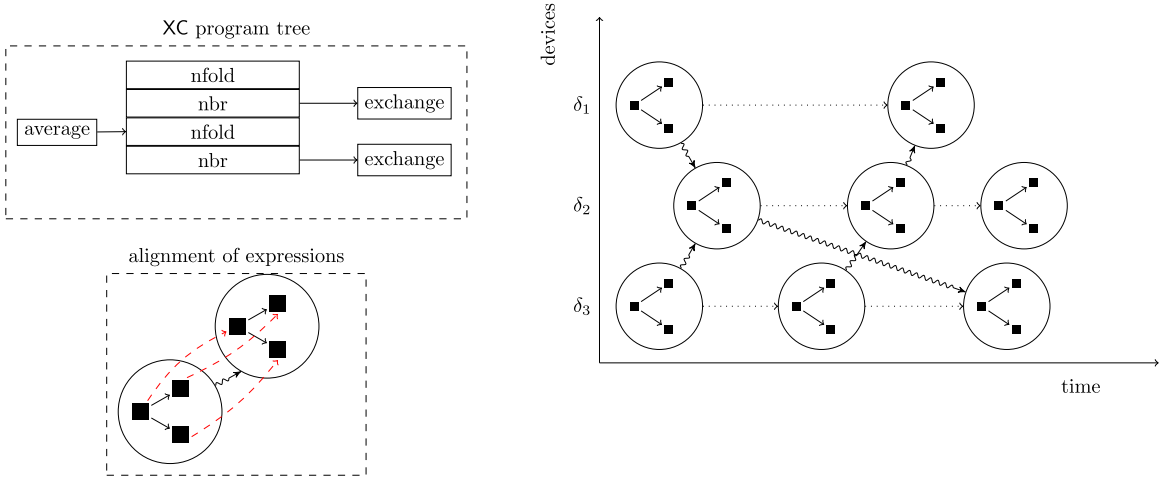


Fig. 3. XC alignment mechanism for Example 4. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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

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

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

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

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

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

2.5. Conditionals

XC supports `if (cond) {e1} else {e2}` conditional expressions. Crucially, their semantics interplays with the communication semantics of XC. Since only the exchange operations in the same position within the AST and stack frame align, with a conditional, an exchange aligns *only* across the devices that take the same branch. Thus, while evaluating an XC sub-expression, we consider only *aligned neighbours*, that are round neighbours which evaluated the same sub-expression (as AST and stack frame). Non-aligned neighbours are never considered in the evaluation of the sub-expression, e.g., for the construction of the $\underline{w}$ 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.

```
1 // type: (bool,bool) → num
2 def distanceToGateways(local, gateway) {
3   if (local) { Infinity } else { distanceTo(gateway) }
4 }
```

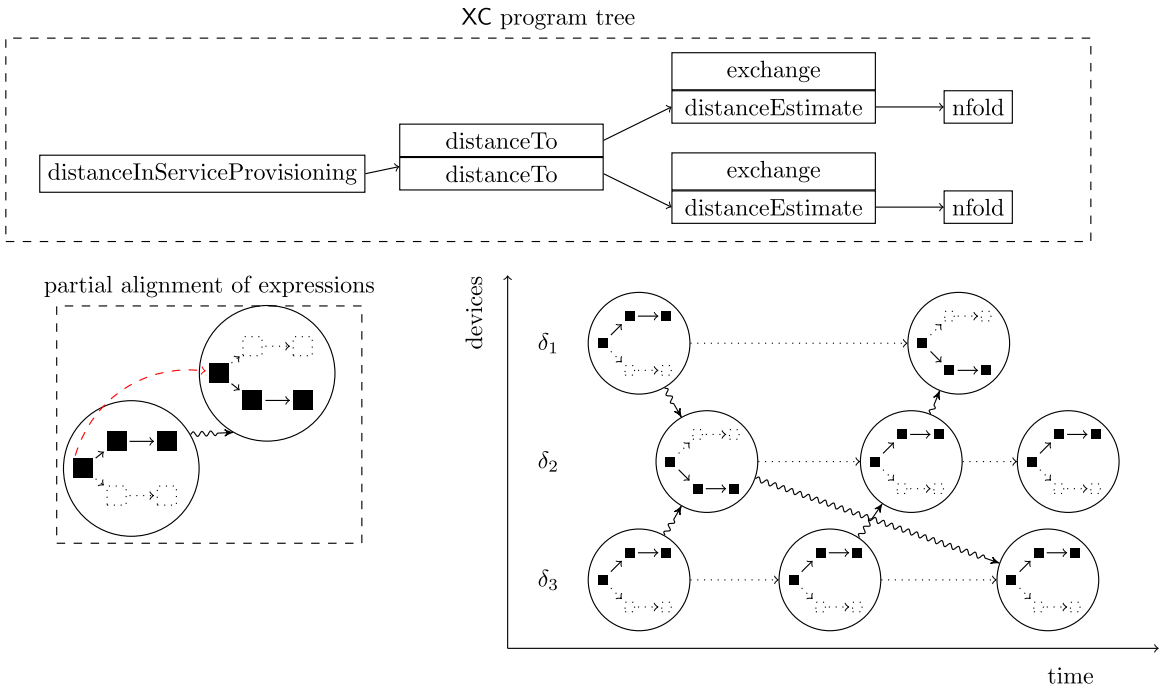


Fig. 4. XC alignment mechanism with conditionals for Example 6.

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

```
1 // type: (bool, bool, bool) → bool
2 def distanceInServiceProvisioning(local,requester,
      gateway){
3   if (local) { distanceTo(requester) }
4   else { distanceTo(gateway) }
5 }
```

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.

```
1  def gossipEver(event) { // type: (bool) → bool
2    exchange(False, (n) => retsend
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 a network of devices realises the expected behaviour, returning true in every device after a button has been pressed anywhere in the network *as soon as possible*, that is, as soon as the fastest chain of messages from the originating event is able to reach the device.

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

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

```
1  def broadcast(dist, value, null) { // 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   )
13 }
```

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

Then, an `exchange` expression sorts out the broadcast received from parent devices, propagating the result to children. The value of the device for the current round is computed in `res` (Line 9), and is taken from the neighbour with the minimum *key*, i.e., minimum rank for a non-`null` value (we assume that `False < True`). For the current device, we use the argument `value` (Line 7). For neighbours, we use the value received from them in `n` (Line 8). The resulting value `res` is returned by exchange and by the whole function, (Line 10). This (possibly band-consuming) value is sent *only* to neighbours which selected the current device as parent, that is, 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.

```
1  // type: (bool, bool, num, 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`.

```
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) → A
8  def collect(dist, radius, value, accumulate,
       extract) {
9    exchange( value, (n) =>
10     val loc = accumulate(n, value); // local
           estimate
11     return loc
12     send extract(loc, normalize(weight(dist,
           radius)))
13   )
14 }
```

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

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

```
1  def accumulate(v, l) { nfold(min, v, l) }
2  def extract(v, w) { mux(w >= 0.25, v, Infinity) }
```

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 *operations centre*. The operations centre might be several hops away from the source, so we want to "broadcast" data hop-by-hop along a short "path" of devices—but without flooding the whole network. The system *(i)* collects and routes data from nodes closer than a certain range towards the closest detector; *(ii)* lets detectors compute the mean levels of warning of the corresponding areas; *(iii)* lets detectors collect and aggregate logs if their mean warning exceeds a certain threshold; and *(iv)* creates self-healing broadcast channels from detectors to the closest operations centres. This logic is implemented by function `smartC` (Fig. 5), which reuses `distanceTo`, `collect`, `broadcast` and `channelBroadcast` (Examples 3 and 8 to 10).

```
1  def smartC(isDetector, isOpsCentre, channelWidth,
       inspectionRadius, commRadius, localWarning,
       warningThreshold, localLog, nullLog, logCat) {
2    val detDist = distanceTo(isDetector);
3    val inspected = detDist < inspectionRadius;
4    val nullReport = (uid, 0, nullLog);
5    val report = if (inspected) {
6      val (sumWarning,numNodes) = collect(detDist,
7        commRadius, (localWarning, 1.0),
8        (v, l) => (nfold(+, fst(v), fst(l)),
9                   nfold(+, snd(v), snd(l))),
10       (v, w) => (fst(v)*w, snd(v)*w)
11     );
12     val meanWarning = sumWarning / numNodes;
13     val localWarning = meanWarning > warningThreshold;
14     val warning = broadcast(detDist,localWarning,False);
15     val logs = if (warning) {
16       collect(detDist,commRadius,localLog,logCat,(v,w)=>v)
17     } else { nullLog };
18     (uid, meanWarning, logs)
19   } else { nullReport };
20   channelBroadcast(isDetector, isOpsCentre, channelWidth,
         report, nullReport)
21 }
```

Fig. 5. XC implementation of a smart city monitoring application.

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).

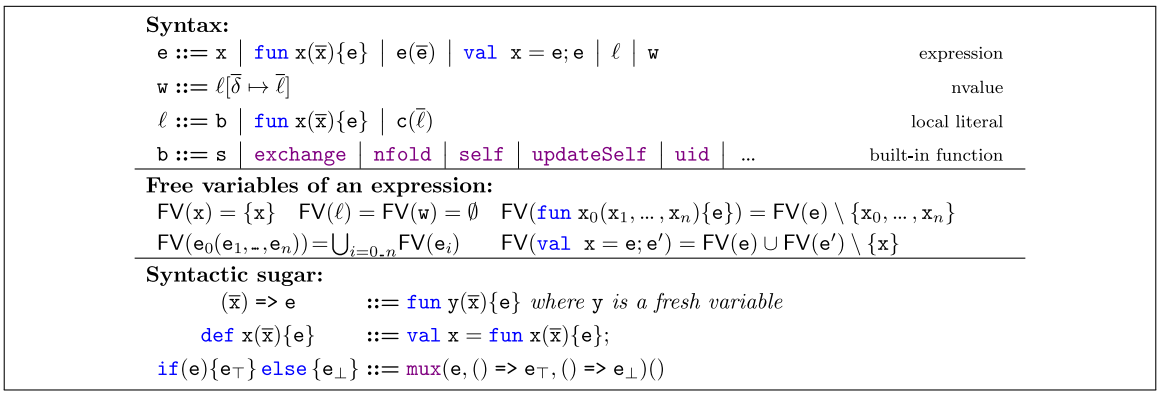


Fig. 6. Syntax (top), free variables (middle) and syntactic sugar (bottom) for FXC expressions.

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 properties are not directly inherited by XC. We further discuss XC expressiveness and resilience properties in Section 8.

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 $x_1, \dots, x_n$ ($n \geq 0$). 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* fun x(x̄){e}, which may have free variables;
- a *function call* e(ē);
- a *let-style* expression val x = e; e;
- a *local literal* ℓ, that is either a built-in function b, a defined function fun x(x̄){e} *without* free variables, or a data constructor c applied to local literals (possibly none);
- an *nvalue* w, as described in Section 2.2.

The set of built-in functions includes *sensors* s, the communication built-in 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 FV(e) = ∅. 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 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* α;
- a (recursive) *data type* $K[\overline{T}]$, consisting of a *parametric type name* K of arity $n \geq 0$, applied to n types $T_1, \dots, T_n$ (possibly zero);
- or a *function type* $(\overline{T}) \to T$.

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 $\mathsf{TV}(T)$ for the set of type variables in T. Polymorphism of functions and data constructors is supported by type schemes TS of the form $\forall\overline{\alpha}.T$ where $\overline{\alpha}$ occur free in T, representing all types obtained by substituting $\overline{\alpha}$ with types $\overline{T}$, as per the type scheme instantiation relation $\prec$. A *typing environment* $\mathcal{A}$ is a set of *assumptions* a : TS where the assumption subject a can be either a variable, a built-in function or constructor. We assume that sensors s have types $() \to T$ where $\mathsf{TV}(T) = \emptyset$. In the typing of programs, an *initial typing environment* $\mathcal{A}_0$ declares a (unique) type scheme for every available constructor and built-in function. This $\mathcal{A}_0$ is then extended with (unique) assumptions for bounded variables encountered while typing sub-expressions of the program. We write $\mathcal{A}$(a) for the unique type scheme a in $\mathcal{A}$.

We specify typing of expressions via judgements $\mathcal{A} \vdash$ e : T which read 'expression e has type T under assumptions $\mathcal{A}$'. Following Igarashi et al. (2001), multiple overbars are expanded together (e.g., $\mathcal{A} \vdash \overline{\text{e}} : \overline{T}$ stands for $\mathcal{A} \vdash \text{e}_1 : T_1, \cdots, \mathcal{A} \vdash \text{e}_n : T_n$). Typing judgements for expressions are syntax-directed. All rules are standard except [T-NVAL], which ensures that all messages $\ell, \overline{\ell}$ of a nvalue have a same type T, and there are no repetitions in $\overline{\delta}$.

Types, type schemes, typing environments and instantiation:

$T ::= \alpha \mid K[\overline{T}] \mid (\overline{T}) \to T$ type — $\mathtt{a} ::= \mathtt{x} \mid \mathtt{b} \mid \mathtt{c}$ assumption subject

$TS ::= \forall \overline{\alpha}.T$ type scheme — $\mathcal{A} ::= \bullet \mid \mathcal{A}, \mathtt{a} : TS$ typing environment

$\forall \overline{\alpha}.T \prec T[\overline{\alpha} := \overline{T}]$ instantiation

Expression typing: $\boxed{\mathcal{A} \vdash \mathtt{e} : T}$

$$\frac{\text{[T-ASS]}\ \mathcal{A}(\mathtt{a}) \prec T \text{ for } \mathtt{a} = \mathtt{x} \text{ or } \mathtt{b}}{\mathcal{A} \vdash \mathtt{a} : T} \qquad \frac{\text{[T-LIT]}\ \mathcal{A}(\mathtt{c}) \prec (\overline{T}) \to T \quad \mathcal{A} \vdash \overline{\ell} : \overline{T}}{\mathcal{A} \vdash \mathtt{c}(\overline{\ell}) : T}$$

$$\frac{\text{[T-NVAL]}\ \mathcal{A} \vdash \ell, \overline{\ell} : T \qquad \overline{\delta} \text{ distinct}}{\mathcal{A} \vdash \ell[\overline{\delta} \mapsto \overline{\ell}] : T} \qquad \frac{\text{[T-APP]}\ \mathcal{A} \vdash \mathtt{e} : (\overline{T}) \to T \qquad \mathcal{A} \vdash \overline{\mathtt{e}} : \overline{T}}{\mathcal{A} \vdash \mathtt{e}(\overline{\mathtt{e}}) : T}$$

$$\frac{\text{[T-FUN]}\ \mathcal{A}, \mathtt{x} : \forall \bullet.(\overline{T}) \to T, \overline{\mathtt{x}} : \forall \bullet.\overline{T} \vdash \mathtt{e} : T}{\mathcal{A} \vdash \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\} : (\overline{T}) \to T} \qquad \frac{\text{[T-VAL]}\ \mathcal{A} \vdash \mathtt{e}_1 : T_1 \qquad \overline{\alpha} = \mathsf{TV}(T_1) \qquad \mathcal{A}, \mathtt{x} : \forall \overline{\alpha}.T_1 \vdash \mathtt{e}_2 : T_2}{\mathcal{A} \vdash \mathtt{val}\ \mathtt{x} = \mathtt{e}_1; \mathtt{e}_2 : T_2}$$

Fig. 7. Typing of FXC expressions.

Auxiliary definitions:

$\sigma ::= \overline{\mathtt{a}} \mapsto \overline{\mathtt{w}}$ where $\mathtt{a} = \mathtt{s}$ or $\mathtt{x}$ — sensor state

$\theta ::= \langle \overline{\theta} \rangle \mid \mathtt{w}\langle \overline{\theta} \rangle$ — value-tree

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

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

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

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

$$\frac{\text{[E-NVAL]}}{\delta; \sigma; \Theta \vdash \mathtt{w} \Downarrow \mathtt{w}; \langle\rangle} \qquad \frac{\text{[E-LIT]}}{\delta; \sigma; \Theta \vdash \ell \Downarrow \ell[]; \langle\rangle} \qquad \frac{\text{[E-VAR]}\ \sigma(\mathtt{x}) = \mathtt{w}}{\delta; \sigma; \Theta \vdash \mathtt{x} \Downarrow \mathtt{w}; \langle\rangle}$$

$$\frac{\text{[E-VAL]}\ \begin{array}{l} \delta; \sigma; \pi_1(\Theta) \vdash \mathtt{e}_1 \Downarrow \mathtt{w}_1; \theta_1 \\ \delta; \sigma; \pi_2(\Theta) \vdash \mathtt{e}_2[\mathtt{x} := \mathtt{w}_1] \Downarrow \mathtt{w}_2; \theta_2 \end{array}}{\delta; \sigma; \Theta \vdash \mathtt{val}\ \mathtt{x} = \mathtt{e}_1; \mathtt{e}_2 \Downarrow \mathtt{w}_2; \langle \theta_1, \theta_2 \rangle} \qquad \frac{\text{[E-APP]}\ \begin{array}{l} \delta; \sigma; \pi_{i+1}(\Theta) \vdash \mathtt{e}_i \Downarrow \mathtt{w}_i; \theta_i \quad \text{for all } i \in 0, \dots, n \\ \delta; \sigma; \pi_{n+2}(\Theta\,|_{\mathtt{f}}) \vdash \mathtt{f}(\mathtt{w}_1, \dots, \mathtt{w}_n) \Downarrow^* \mathtt{w}_{n+1}; \theta_{n+1} \text{ where } \mathtt{f} = \mathtt{w}_0(\delta) \end{array}}{\delta; \sigma; \Theta \vdash \mathtt{e}_0(\mathtt{e}_1, \dots, \mathtt{e}_n) \Downarrow \mathtt{w}_{n+1}; \mathtt{f}[]\langle \theta_0, \dots, \theta_{n+1} \rangle}$$

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

$$\frac{\text{[A-FUN]}\ \delta; \sigma; \Theta \vdash \mathtt{e}[\mathtt{x} := \mathtt{fun}^{\tau}\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}, \overline{\mathtt{x}} := \overline{\mathtt{w}}] \Downarrow \mathtt{w}; \theta}{\delta; \sigma; \Theta \vdash \mathtt{fun}^{\tau}\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}(\overline{\mathtt{w}}) \Downarrow^* \mathtt{w}; \theta} \qquad \frac{\text{[A-XC]}\ \begin{array}{l} \Theta = \overline{\delta} \mapsto \overline{\mathtt{w}}\langle \dots \rangle \quad \mathtt{w} = \mathtt{w}_i[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)] \\ \delta; \sigma; \pi_1(\Theta) \vdash \mathtt{w}_f(\mathtt{w}) \Downarrow (\mathtt{w}_r, \mathtt{w}_s); \theta \end{array}}{\delta; \sigma; \Theta \vdash \mathtt{exchange}(\mathtt{w}_i, \mathtt{w}_f) \Downarrow^* \mathtt{w}_r; \mathtt{w}_s\langle \theta \rangle}$$

$$\frac{\text{[A-SENS]}\ \sigma(\mathtt{s}) = \mathtt{w}}{\delta; \sigma; \Theta \vdash \mathtt{s}() \Downarrow^* \mathtt{w}; \langle\rangle} \qquad \frac{\text{[A-UID]}}{\delta; \sigma; \Theta \vdash \mathtt{uid}() \Downarrow^* \delta; \langle\rangle} \qquad \frac{\text{[A-SELF]}}{\delta; \sigma; \Theta \vdash \mathtt{self}(\mathtt{w}) \Downarrow^* \mathtt{w}(\delta); \langle\rangle}$$

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

Fig. 8. Device (big-step) operational semantics of FXC.

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 $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \mathtt{w}; \theta$, to be read as "expression $\mathtt{e}$ evaluates to nvalue $\mathtt{w}$ and value-tree θ on device δ with respect to sensor values σ and value-tree environment Θ", where:

- $\mathtt{w}$ is called the *result* of $\mathtt{e}$;
- θ 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 $\mathtt{e}$, and the stack frames of function calls;
- Θ collects the (non expired) value-trees received by the most recent firings of neighbours of δ, as a map $\delta_1 \mapsto \theta_1$, ..., $\delta_n \mapsto \theta_n$ ($n \geq 0$) from device identifiers to value-trees;
- σ associates nvalues to every sensor name $\mathtt{s}$, and in case of open expression, to every free variable $\mathtt{x}$ occurring in $\mathtt{e}$.[2]

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.

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

The syntax of value-trees and value-tree environments is in Fig. 8 (top). The rules for judgement $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \mathtt{v}; \theta$ (Fig. 8, middle)

[2] Defining 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 $\mathtt{e}'$ of $\mathtt{e}$ w.r.t. the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ. This *alignment* process exploits the auxiliary "projection" functions π_i for any positive natural number i (Fig. 8, top). When applied to a value-tree θ, π_i returns the ith sub-tree θ_i of θ. When applied to a value-tree environment Θ, π_i acts pointwise on the value-trees in Θ.

The alignment process ensures that value-trees in environment Θ always correspond to the evaluation of the same sub-expression currently being evaluated. To ensure this match holds (as said before, of the stack frame and position in the AST), in the evaluation of a function application $\mathtt{f}(\overline{\mathtt{w}})$, the environment Θ is reduced to the smaller set $\Theta|_{\mathtt{f}}$ of trees which corresponded to the evaluation of a function with the same *name* (as defined in Fig. 8 (top)).

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

Rule [E-APP] is mostly standard eager function application: the function expression $\mathtt{e}_0$ and each argument $\mathtt{e}_i$ are evaluated w.r.t. $\pi_{i+1}(\Theta)$ producing result $\mathtt{v}_i$ and value-tree θ_i. Then, the function application itself is demanded to the auxiliary evaluation rules, w.r.t. the last sub-tree of the trees corresponding to the same function: $\pi_{n+2}(\Theta|_{\mathtt{f}})$. Notice that the reduction $\Theta|_{\mathtt{f}}$ ensures that only trees corresponding the same function execution (stack frame) are considered. The auxiliary rule [A-FUN] handles the application of fun-expression, which evaluates the body after replacing the arguments $\overline{\mathtt{x}}$ with their provided values $\overline{\mathtt{w}}$, and the function name $\mathtt{x}$ with the fun-expression itself. Rules [A-UID] and [A-SELF] trivially encode the behaviour of the `uid` and `self` built-ins. Rule [A-SENS] defines the behaviour of sensors through the values stored in σ. Rule [A-XC] evaluates an exchange-expression, realising the behaviour described at the beginning of Section 2.3. Notation $\mathtt{w}_1[\overline{\delta} \mapsto \overline{\mathtt{w}}(\delta)]$ is used to represent the nvalue $\mathtt{w}_1$ after the update for each i of the message for δ_i with the custom message $\mathtt{w}_i(\delta)$. The result is fed as argument to function $\mathtt{w}_f$: the first element of the resulting pair is the overall result, while the second is used to tag the root of the value-tree. Rule [A-FOLD] encodes the `nfold` operators. First, the domain of Θ is inspected, giving a (sorted) list $\delta_1, \ldots, \delta_n$. An initial local value ℓ_0 is set to the "self-message" of the third argument. Then, a sequence of ℓ_i is defined, each by applying function $\mathtt{w}_1$ to the previous element in the sequence and the value $\mathtt{w}_2(\delta_i)$ (skipping δ itself). The final result ℓ_n is the result of the application, with empty value-tree. Auxiliary rules for the other available built-in functions are standard, do not depend on the environment, hence have been omitted.

Network-level semantics. The evolution of a network executing a program $\mathtt{e}_{\mathtt{main}}$ 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 $\rightsquigarrow$. This idea can be formalised through *augmented event structures*, which augments 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* $\mathbb{E} = \langle E, \rightsquigarrow, d, s\rangle$ is a tuple where:

- E is a countable set of *events* ϵ,
- $\rightsquigarrow \subseteq E \times E$ is a *messaging* relation,
- $d : E \to \Delta$ is a mapping from events to the devices where they happened, and
- $s : E \to S$ maps each event ϵ to a *sensors status* σ (as in the device-level semantics),

such that:

- $\epsilon_1 = \epsilon_2$ whenever $d(\epsilon_1) = d(\epsilon_2)$ and $\epsilon_i \rightsquigarrow \epsilon$ for $i = 1, 2$ (i.e., all predecessors of an event happened on different devices);
- there are no sequences $\epsilon_1 \rightsquigarrow \ldots \rightsquigarrow \epsilon_n \rightsquigarrow \epsilon_1$ (i.e., the $\rightsquigarrow$ relation is acyclic);
- the set $X_\epsilon = \{\epsilon' \in E \mid \epsilon' \rightsquigarrow \ldots \rightsquigarrow \epsilon\}$ of events that can reach ϵ in $\rightsquigarrow$ is finite for all ϵ (i.e., $\rightsquigarrow$ is well-founded and locally finite).

We say that event ϵ' is a *supplier* of event ϵ iff $\epsilon' \rightsquigarrow \epsilon$. We call *causality relation* the irreflexive partial order $< \subseteq E \times E$ obtained as the transitive closure of $\rightsquigarrow$.

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 ϵ, into sub-spaces: the "causal past", "causal future", and "present" (concurrent). A computation at ϵ 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 $\rightsquigarrow$-connected events associated to the same device. Communication is captured by $\rightsquigarrow$-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 $\mathbb{E} = \langle E, \rightsquigarrow, d, s\rangle$ be an augmented event structure and $\mathbf{V}_T$ be the domain of nvalues of type T. A *space–time value* (in $\mathbb{E}$ of type T) is a function mapping events to nvalues $\Phi : E \to \mathbf{V}_T$.

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

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

Let $\theta^{\mathbb{E}}_{\mathtt{e}} : E \to \Omega$ (where Ω is the set of all value-trees) and $\Phi^{\mathbb{E}}_{\mathtt{e}} : E \to \mathbf{V}_T$ be defined by induction on ϵ in E, so that $d(\epsilon); s(\epsilon); \Theta_\epsilon \vdash \mathtt{e} \Downarrow \Phi^{\mathbb{E}}_{\mathtt{e}}(\epsilon); \theta^{\mathbb{E}}_{\mathtt{e}}(\epsilon)$ where $\Theta_\epsilon = \{d(\epsilon') \mapsto \theta^{\mathbb{E}}_{\mathtt{e}}(\epsilon') : \epsilon' \rightsquigarrow \epsilon\}$. Then we say that $\Phi^{\mathbb{E}}_{\mathtt{e}} : E \to \mathbf{V}_T$ is the evaluation of expression $\mathtt{e}$ on $\mathbb{E}$.

Notice that this semantics models round computations ϵ that are performed with respect to a given Θ_ϵ that cannot change during the local execution.[3] Furthermore, notice that a program may not have *any* interpretation in an event structure $\mathbb{E}$ (i.e., the definition may fail) in case the program was non-terminating in some ϵ in $\mathbb{E}$. 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.

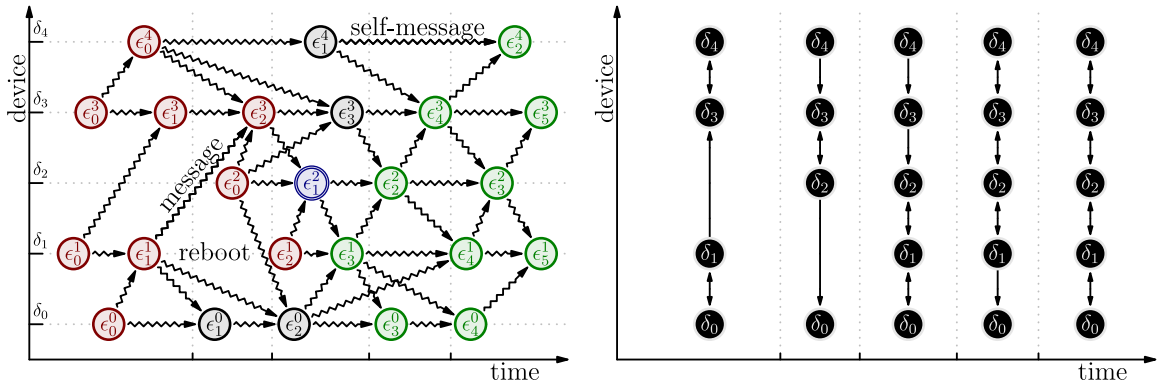


Fig. 9. Left: An augmented event structure $\mathbb{E} = \langle E, \rightsquigarrow, d, s\rangle$ where $\mathbb{E} = \{\epsilon_0^0, \dots, \epsilon_4^0, \epsilon_0^1, \dots, \epsilon_5^1, \epsilon_0^2, \dots, \epsilon_4^2, \epsilon_0^3, \dots, \epsilon_3^3, \epsilon_0^4, \dots, \epsilon_5^4\}$ consists of 24 events such that $d(\epsilon_j^i) = \delta_i$; it depicts events (circle nodes), messaging relations (curly arrows), devices $\delta_0, \dots, \delta_4$ (y-axis) and each circle node is labelled with the depicted event of $\mathbb{E}$. Colours denote the causal relation w.r.t. the reference event ϵ_1^2 (doubly-circled, blue), partitioning events into causal past (red), causal future (green) and concurrent (non-ordered, in black). **Right:** A (possible) evolving network topology on devices which can give rise to $\mathbb{E}$ (assuming some selected messages to be dropped). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

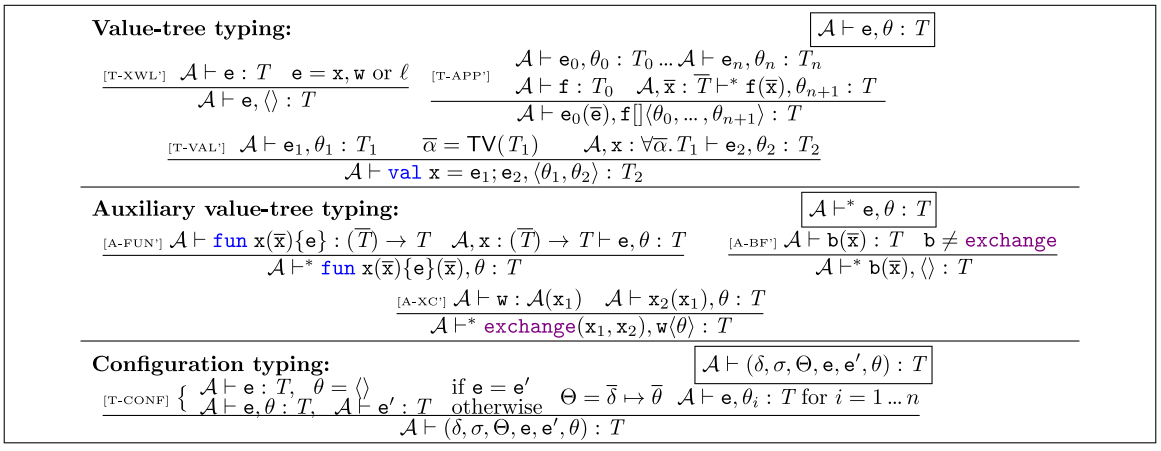


Fig. 10. FXC: value-tree and configuration typing.

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 $\mathtt{e}_{\mathtt{main}}$ 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 $\mathtt{e}_{\mathtt{main}}$ is a procedure that, when executed in a round ϵ, takes as input the current device identifier δ, the sensor status σ, and the *environment* Θ grouping the message-collections that δ received from the suppliers of event ϵ, and outputs a value $\mathtt{w} \in \mathbf{V}$ (the *result* of $\mathtt{e}_{\mathtt{main}}$) and the *message-collection* θ to send to neighbours in return.

In the *global viewpoint*, given by the network-level semantics, the program $\mathtt{e}_{\mathtt{main}}$ describes a partial functor which, when applied to an augmented event structure $\mathbb{E} = \langle E, \rightsquigarrow, d, s\rangle$,[4] returns a space–time value Φ encoding the values produced by the program across space–time.

[4] The function is partial since a program has no interpretation in an $\mathbb{E}$ with events ϵ in which P diverges.

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 $\mathcal{A} \vdash \mathtt{e}, \theta : T$, ensures that both $\mathcal{A} \vdash \mathtt{e} : T$ holds, and θ has a structure that could possibly be produced by an evaluation of $\mathtt{e}$. The latter, formalised by judgement $\mathcal{A} \vdash (\delta, \sigma, \Theta, \mathtt{e}, \mathtt{e}', \theta) : T$, ensures that every value-tree in Θ could be an outcome of $\mathtt{e}$, and either θ is empty and $\mathtt{e} = \mathtt{e}'$ (initial configuration) or $\mathtt{e}', \theta$ could be an outcome of $\mathtt{e}$ (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 ∞ to model non-terminating executions. Initial configurations represent computations that still need to be done, and are thus fully determined by the left-hand-side of the big-step judgement, with the right-hand side replaced by dummy values for convenience of definitions (a duplicate of the expression, and the empty value-tree). This choice of defaults is made so that the only initial configurations $c = (\delta, \sigma, \Theta, \mathtt{e}, \mathtt{e}', \theta)$ that are also final configurations are those with $\mathtt{e} = \mathtt{e}' = \mathtt{w}, \theta = \langle\rangle$, and represent

the evaluation of a value (which results in itself). We write $c \Downarrow r$ 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 $c = (\delta, \sigma, \Theta, \mathtt{e}, \mathtt{e}, \langle\rangle)$ be such that $\mathcal{A} \vdash c : T$ holds, σ include values for every sensor and variable as per assumptions $\mathcal{A}$, and assume that $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \mathtt{w}; \theta$ holds for some $\mathtt{w}, \theta$. Then $\mathcal{A} \vdash \mathtt{w} : T$ and $\mathcal{A} \vdash \mathtt{e}, \theta : T$ hold.*

Proof. We prove the result by induction on the height of tree θ. For expressions $\mathtt{e}$ composed of multiple sub-expressions $\mathtt{e}_1, \dots, \mathtt{e}_n$ we define sub-configurations $c_i = (\delta, \sigma, \pi_i(\Theta), \mathtt{e}_i, \mathtt{e}_i, \langle\rangle)$ for $i \leq n$, together with their evaluation results $\mathtt{w}_i, \theta_i$ (where applicable).

- $\mathtt{e} = \mathtt{x}$ or $\mathtt{w}$ or ℓ: By either rule [E-VAR], [E-NVAL] or [E-LIT], $\mathtt{e}$ evaluates to $\mathtt{w}, \langle\rangle$ where $\mathtt{w}$ is respectively $\sigma(\mathtt{e})$, $\mathtt{e}$ itself, or $\mathtt{e}[]$. All of them trivially have type T by hypothesis on σ and rule [T-XWL']. Notice that this case includes $\mathtt{e} = \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$, since by hypothesis on σ every free variable in $\mathtt{e}$ has to be bound in σ.
- $\mathtt{e} = \mathtt{val}\ \mathtt{x} = \mathtt{e}_1; \mathtt{e}_2$: 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] so does c.
- $\mathtt{e} = \mathtt{e}_0(\overline{\mathtt{e}})$: By rule [E-APP], [T-APP'] and [T-CONF], c_i has type T_i for $i \leq n$. By rule [T-CONF], Θ must consist of value-trees that can be produced by the evaluation of $\mathtt{e}$. This ensures that in the root of each of those value-trees there is a functional value $\mathtt{f}[]$, and the corresponding sub-tree must be coherent with the application $\mathtt{f}(\overline{\mathtt{x}})$. By rule [E-APP], the evaluation of $\mathtt{f}(\overline{\mathtt{w}})$ is carried out with respect to $\pi_{n+2}(\Theta|_{\mathtt{f}})$, which only contains trees that are compatible with $\mathtt{f}(\overline{\mathtt{x}})$. Thus, we can apply the inductive hypothesis to obtain that $c_{n+1} = (\delta, \sigma, \pi_{n+2}(\Theta|_{\mathtt{f}}), \mathtt{f}(\overline{\mathtt{w}}), \langle\rangle)$ has type T for $\overline{\mathtt{w}}$ of type $\overline{T}$. The thesis follows by rule [T-APP'] and [T-CONF].
- $\mathtt{e} = \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_1\}(\overline{\mathtt{w}})$: By rule [A-FUN], $\mathtt{e}$ evaluates to $\mathtt{e}_1[\mathtt{x} := \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_1\}, \overline{\mathtt{x}} := \overline{\mathtt{w}}]$, and the thesis follows by rule [A-FUN'] and [T-CONF].
- $\mathtt{e} = \mathtt{b}(\overline{\mathtt{w}})$: If $\mathtt{b}$ is not `exchange`, the thesis follows trivially by rule [A-BF'] and either [A-SENS], [A-UID], [A-SELF] or [A-FOLD]. Assume now that $\mathtt{e} = \mathtt{exchange}(\mathtt{w}_1, \mathtt{w}_2)$. Following rule [A-XC], we construct $\mathtt{w}$ by picking values from Θ, which must have the correct type by rule [T-CONF]. The overall result $\mathtt{w}_r, \theta$ is then obtained as the result of application $\mathtt{w}_2(\mathtt{w})$, 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 $\mathtt{e}$ on an augmented event structure $\mathbb{E}$ only contains value-tree environments that are well-formed for $\mathtt{e}$. 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 ∞ modelling divergence, with corresponding judgements $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \infty$. 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 $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \infty$ can be derived with a finite proof tree, and conversely, only judgements $\delta; \sigma; \Theta \vdash \mathtt{e} \Downarrow \infty$ can be derived with an infinite proof tree (see Ancona et al., 2018 for further details).

Theorem 2 (*Soundness-May*). *Let $c = (\delta, \sigma, \Theta, \mathtt{e}, \mathtt{e}, \langle\rangle)$ be a well-typed initial configuration in* XC. *Then there exist a final configuration r (possibly ∞) such that $c \Downarrow r$.*

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 $r \neq \infty$ such that $c \Downarrow r$. Given any $c = (\delta, \sigma, \Theta, \mathtt{e}, \mathtt{e}, \langle\rangle) \in WT^\infty$, then $c \Downarrow \infty$ is the consequence of a rule where, for all premises of shape $c' \Downarrow \infty$, $c' \in WT^\infty$, and all premises of shape $c' \Downarrow r$ with $r \neq \infty$ are derivable.*

Proof. The proof is by case analysis over $\mathtt{e}$. Let T be the type of c, and whenever $\mathtt{e}$ has sub-expressions $\mathtt{e}_1, \dots, \mathtt{e}_n$ we consider $c_i = (\delta, \sigma, \pi_i(\Theta), \mathtt{e}_i, \mathtt{e}_i, \langle\rangle)$ for $i \leq n$.

- $\mathtt{e} = \mathtt{x}, \mathtt{w}$ or ℓ: empty case since such expressions $\mathtt{e}$ always evaluate to a configuration $r \neq \infty$ by rules [E-VAR], [E-NVAL] or [E-LIT].
- $\mathtt{e} = \mathtt{val}\ \mathtt{x} = \mathtt{e}_1; \mathtt{e}_2$: By rule [E-VAL], it must be that either $\mathtt{e}_1$ does not evaluate to $r \neq \infty$ or $\mathtt{e}_2[\mathtt{x} := \mathtt{v}_1]$ does not evaluate to $r \neq \infty$. The thesis follows by either rule [D-VAL1] or [D-VAL2] respectively.
- $\mathtt{e} = \mathtt{e}_0(\overline{\mathtt{e}})$: by rule [T-APP'], we have that c_i has type T_i and $\mathtt{f}$ has type scheme $TS \prec (\overline{T}) \to T$. We have the following sub-cases:
  - $\exists k$ such that $\nexists r_k \neq \infty$ such that $c_k \Downarrow r_k$: let j be least with this property, then $c \Downarrow \infty$ by rule [D-APP1], with derivable premises $c_i \Downarrow r_i$ for $i < j$ and premise $c_j \Downarrow \infty$ such that $c_j \in WT^\infty$;
  - $c_i \Downarrow r_i = (\delta, \sigma, \pi_i(\Theta), \mathtt{w}_i, \theta_i)$ for all $i \leq n$, and $\nexists r_{n+1} \neq \infty$ such that $c_{n+1} = (\delta, \sigma, \pi_{n+2}(\Theta|_{\mathtt{f}}), \mathtt{f}(\overline{\mathtt{w}}), \langle\rangle) \Downarrow r_{n+1}$: then $c \Downarrow \infty$ by rule [D-APP2], with derivable premises $c_i \Downarrow r_i$ for $i < n$ and premise $c_j \Downarrow \infty$ such that $c_j \in WT^\infty$;
  - $c_i \Downarrow r_i$ for all $i \leq n+1$: empty case since by rule [E-APP] there is r such that $c \Downarrow r$.
- $\mathtt{e} = \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_1\}(\overline{\mathtt{w}})$: By rule [A-FUN], it must be the case that $\mathtt{e}_1[\mathtt{x} := \mathtt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_1\}]$ does not evaluate to $r \neq \infty$. Then c is consequence of rule [D-FUN] with premise in WT^∞.
- $\mathtt{e} = \mathtt{b}(\overline{\mathtt{w}})$: By rules [A-SENS], [A-UID] and [A-SELF] we have that $\mathtt{b}$ cannot be a sensor or `uid`, `self` as such applications always evaluate to a configuration $r \neq \infty$. If $\mathtt{b}$ is `nfold` and c does not evaluate to an $r \neq \infty$, there must be a minimal $j \leq n$ for which the corresponding premise is not derivable, thus in WT^∞. The thesis follows by rule [D-FOLD].
  Assume now that $\mathtt{e} = \mathtt{exchange}(\mathtt{w}_1, \mathtt{w}_2)$. By rule [T-XC'], we have that c_1 has type T. By rule [A-XC] we have the following sub-cases:
  - $\nexists r_1 \neq \infty$ such that $c_1 \Downarrow r_1$: then $c \Downarrow \infty$ is the consequence of rule [D-XC] with premise $c_1 \Downarrow \infty$ such that $c_1 \in WT^\infty$;

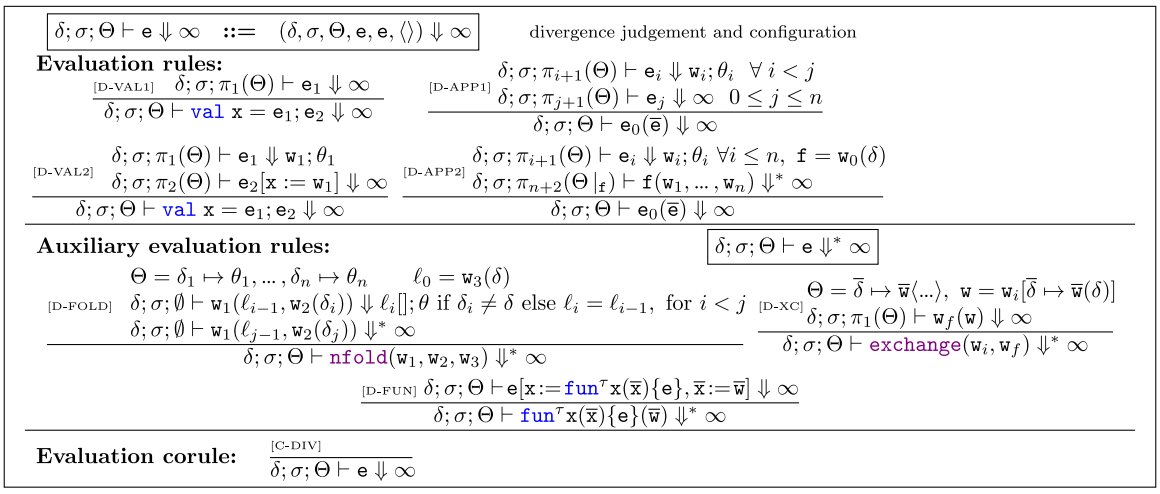


Fig. 11. FXC: divergence operational semantics.

- $c_1 \Downarrow r_1 = (\delta, \sigma, \pi_1(\Theta), \mathtt{w}_1, \theta_1)$ and $\nexists r_2 \neq \infty$ such that $c_2 = (\delta, \pi_2(\Theta), \mathtt{w}_2(\mathtt{w}), \langle\rangle) \Downarrow r_2$: since c has type T, by rules [T-XC'] and [T-CONF] it follows that every value-tree in Θ as a value of type T in its root. Similarly, c_1 has type T hence r_1 has type T by Theorem 1 hence $\mathtt{w}_1$ has type T. It follows that $\mathtt{w}$ in rule [A-XC] has type T by rule [T-NVAL], hence c_2 has type T by rule [T-XC'] and substitution. Then $c \Downarrow \infty$ is the consequence of rule [D-XC] with derivable premise $c_1 \Downarrow r_1$ and premise $c_2 \Downarrow \infty$ such that $c_2 \in WT^\infty$;
- $c_1 \Downarrow r_1$ and $c_2 \Downarrow r_2$: empty case since by rule [A-XC] it would imply that $\exists r$ such that $c \Downarrow r$. □

We are now able to prove that the execution of a round of computation in a local device is deterministic (although the execution of a whole system is not).

Theorem 4 (*Round Determinism*). *Let $c = (\delta, \sigma, \Theta, \mathtt{e}, \mathtt{e}, \langle\rangle)$ be a well-typed initial configuration in* XC. *Then there exist a unique final configuration r (possibly ∞) such that $c \Downarrow r$.*

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

As a consequence of determinism, the soundness-may result in Theorem 2 (which ensures that there *exists* a non-stuck computation, 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 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 ℓ that is not dependent from neighbours, while a neighbouring value represents a neighbour-dependent value, as a map $\phi = \overline{\delta} \mapsto \overline{\ell}$ from local values to neighbours. Neighbouring values are used to model received messages, while only local values are allowed for messages to send, thus not supporting differentiated messages to neighbours.

XC combines these two categories into a single class of nvalues $\mathtt{v} = \ell[\overline{\delta} \mapsto \overline{\ell}]$. In particular, *local values* are equivalent to nvalues $\ell[]$ without custom messages, and *neighbouring values* are equivalent to nvalues with any valid default message; while preserving the behaviour of FC programs interpreted within XC. This unification allows a simpler type system and for differentiated messages to neighbours, increasing the expressiveness of the language.

By interpreting FC values as nvalues, all FC message-exchanging constructs (`nbr`, `rep` Viroli et al., 2018 and `share` Audrito et al., 2020) can be modelled within XC: `nbr` is the same defined function introduced in Section 2.2, just restricted to operate on local values only; `share` corresponds to an `exchange` with `retsend` restricted to operate on local values only; and $\mathtt{rep}(\mathtt{e}_1)\{(\mathtt{x}) \Rightarrow \mathtt{e}_2\}$ can be translated to $\mathtt{exchange}(\mathtt{e}_1, (\mathtt{x}) \Rightarrow \mathtt{retsend}\ \mathtt{e}_2[\mathtt{x} := \mathtt{self}(\mathtt{x})])$. Notice that the converse translation is not possible, as `nbr`, `rep` or `share` expressions with arguments of neighbouring type have no defined behaviour in FC. Thus, `nbr` and `exchange` in XC are strictly more expressive than their corresponding FC counterparts `nbr` and `share`: they can be used with expressions producing nvalues with custom messages to model differentiated messages.

5.3. Self-stabilisation

While the interpretation of a general program has to be given in spatio-temporal terms, self-stabilising programs allow for a space-only representation, where *augmented event structures* and *space–time values* are replaced by *network graphs* and *computational fields*.

Definition 4 (*Computational Field*). A *network graph* $\mathbf{G} = \langle \Delta, \rightarrowtail, \Sigma \rangle$ is a finite set of devices Δ with a reflexive neighbouring relation $\rightarrowtail \subseteq \Delta \times \Delta$ and a sensors-map $\Sigma : \Delta \to S$. A *computational field* in $\mathbf{G}$ and $\mathbf{V}_T$ is a map $\Psi : \Delta \to \mathbf{V}_T$ where $\mathbf{V}_T$ is the domain of values of type T.

Intuitively, $\delta_1 \rightarrowtail \delta_2$ (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 $\epsilon_1 \rightsquigarrow \epsilon_2$ for events ϵ_i

```
es   ::=  x | fun x(x̄){es} | es(ēs) | val x = es; es | ℓ_s | w                 self-stabilising expression
        | nbr(es, es)
        | exchange(e, (x) => retsend f^C(x, es, ē))
        | exchange(e, (x) => retsend es(mux(nbrlt(es), x, es), ēs))
        | exchange(e, (x) => retsend f^R(nfold(min, f^MP(x, ēs), es), self(x), ē))
ℓ_s  ::=  fun x(x̄){es} | c(ℓ̄_s) | s | nfold | self | uid ...                 exchange-free local literal
```

Fig. 12. Syntax of the self-stabilising fragment of XC. Self-stabilising expressions `es` occurring inside `exchange` statements cannot contain free occurrences of the `exchange`-bound variable `x`. Function `nbrlt` is defined as `def nbrlt(x) { nbr(x, self(x)) < self(x) }`.

on devices δ_i ($1 \leq i \leq 2$). Note $\rightarrowtail$ 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 $\mathbf{G}$, for any $\mathbb{E}$ which eventually conforms to $\mathbf{G}$, the result of the program eventually becomes a unique computational field $\psi = [\![\mathtt{e}]\!]^{\text{self-stab}}(\mathbf{G})$ depending only on $\mathbf{G}$, and independent of the specific $\mathbb{E}$ considered. Thus, a self-stabilising program can always adapt to a new input (the eventual $\mathbf{G}$) given enough time, regardless of its previous history (the specific $\mathbb{E}$ before conforming to $\mathbf{G}$).

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

Definition 5 (*Stabilising Augmented Event Structure and Limit*)**.** Let $\mathbb{E} = \langle E, \rightsquigarrow, d, s\rangle$ be a countably infinite augmented event structure. We say that $\mathbb{E}$ is *stabilising* to its limit $\mathbf{G} = \langle \Delta, \rightarrowtail, \Sigma\rangle = \lim \mathbb{E}$ iff $\Delta = \{\delta \mid \{\epsilon \in E.\ d(\epsilon) = \delta\}$ is infinite$\}$ is the set of devices appearing infinitely often in $\mathbb{E}$, and for all except finitely many $\epsilon \in E$, $s(\epsilon) = \Sigma(d(\epsilon))$ and the devices of suppliers are the neighbours of the device of ϵ: $\{d(\epsilon') \mid \epsilon' \rightsquigarrow \epsilon\} = \{\delta' \mid \delta' \rightarrowtail d(\epsilon)\}$.

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 $\mathbb{E}$ may not exist (thus $\mathbb{E}$ not being stabilising), if it does it is unique, as the definition above provides a constructive procedure for Δ, Σ and $\rightarrowtail$ in terms of what appears infinitely often in $\mathbb{E}$.

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 stabilisation) can also be applied to space–time values and expressions as shown in the following.

Definition 6 (*Stabilising Space-Time Value and Limit*)**.** Let Φ be a space–time value on a stabilising augmented event structure $\mathbb{E} = \langle E, \rightsquigarrow, d, s\rangle$ with limit $\mathbf{G} = \lim \mathbb{E}$. We say that Φ is stabilising to its limit $\Psi = \lim \Phi$ iff for all except finitely many $\epsilon \in E$, $\Phi(\epsilon) = \Psi(d(\epsilon))$.

Notice that $\mathbf{G}$ is not a parameter of the definition above, by being uniquely determined by $\mathbb{E}$. These definitions induce the notion of self-stabilisation of a program, interpreted as a procedure producing a space–time value from an augmented event structure.

Definition 7 (*Self-Stabilising Program and Limit*)**.** Let `e` be an expression, which given any augmented event structure $\mathbb{E}$ produces a space–time value Φ. We say that `e` is *self-stabilising* iff for any $\mathbb{E}$ with limit $\mathbf{G}$, the corresponding space–time value Φ has a limit Ψ.

An extension of the self-stabilising fragment in Viroli et al. (2018) using `exchange` is given in Fig. 12, defining a class `es` of self-stabilising expressions, which may be:

- 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 $\mathtt{f}(\mathtt{w}_o, \mathtt{w}, \overline{\mathtt{v}})$ is C with respect to a metric $\mathrm{dist}(\ell_1, \ell_2)$ iff its return value $\mathtt{w}_r$ has lower maximum distance from `w` than $\mathtt{w}_o$: $\max_\delta \mathrm{dist}(\mathtt{w}_r(\delta), \mathtt{w}(\delta)) < \max_\delta \mathrm{dist}(\mathtt{w}_o(\delta), \mathtt{w}(\delta))$.

Monotonic non-decreasing (M) a stateless[5] function $\mathtt{f}(\mathtt{x}, \overline{\mathtt{x}})$ is M iff it applies pointwise on nvalues and whenever $\ell_1 \leq \ell_2$, also $\mathtt{f}(\ell_1, \overline{\mathtt{v}}) \leq \mathtt{f}(\ell_2, \overline{\mathtt{v}})$.

Progressive (P) a stateless function $\mathtt{f}(\mathtt{x}, \overline{\mathtt{x}})$ is P iff it applies pointwise on nvalues and $\mathtt{f}(\ell, \overline{\mathtt{v}}) > \ell$ or $\mathtt{f}(\ell, \overline{\mathtt{v}}) = \top$ (where $\top$ denotes the unique maximal element of the relevant type).

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

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 $\mathtt{f}^{\mathsf{C}}$ and a self-stabilising value `es`. The function $\mathtt{f}^{\mathsf{C}}$ may also have additional (not necessarily self-stabilising) inputs $\overline{\mathtt{e}}$. If the range of the metric granting convergence of $\mathtt{f}^{\mathsf{C}}$ is a well-founded set[6] 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 neighbourhood'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 $\mathtt{f}(\overline{\mathtt{x}})$ 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 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 $\mathtt{f}^{\mathsf{MP}}$ (possibly depending also on other self-stabilising inputs). As specified above, $\mathtt{f}^{\mathsf{MP}}$ needs to operate pointwise on nvalues.

Afterwards, the minimum among those values and a local self-stabilising value is then selected by function nfold. Finally, this minimum is fed to the *raising* function $\mathtt{f}^{\mathsf{R}}$ together with the value of self(x) (and possibly any other inputs $\overline{\mathtt{e}}$), obtaining a result that is higher than at least one of the two parameters. We assume that the partial orders $<, \lhd$ 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 ℓ = self(es) passed as third argument of nfold, after being increased by $\mathtt{f}^{\mathsf{MP}}$ 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 $\mathbb{E} = \langle E, \rightsquigarrow, <, d\rangle$ 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, 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 val -expression $\mathtt{val\ x = es_1; es_2}$. By inductive hypothesis, the sub-expression $\mathtt{es}_1$ stabilises to Ψ after a certain event. Then, $\mathtt{val\ x = es_1; es_2}$ evaluates to the same value as the expression $\mathtt{es}_2$ after adding Ψ to Σ, which is self-stabilising by inductive hypothesis. Thus, the whole val -expression self-stabilises.
- A functional application $\mathtt{es}(\overline{\mathtt{es}})$. By inductive hypothesis, all sub-expressions self-stabilise to $\Psi, \overline{\Psi}$ after a certain event. First, assume that $\Psi(\delta)(\delta) = \mathtt{f}$ is a built-in (which cannot be an exchange by the posed syntax restrictions) or sensor. Then $\mathtt{f}(\overline{\mathtt{es}})$ stabilises immediately as built-ins and sensors do not depend on the environment Θ. Otherwise, f has to be a fun -expression $\mathtt{fun\ x}(\overline{\mathtt{x}})\ \{\mathtt{es}'\}$, and thus $\mathtt{f}(\overline{\mathtt{es}})$ evaluates to the same value as es′ after adding $\mathtt{x} \mapsto \Psi, \overline{\mathtt{x}} \mapsto \overline{\Psi}$ to Σ, which is self-stabilising by inductive hypothesis.
- A neighbourhood pattern $\mathtt{nbr}(\mathtt{es}_1, \mathtt{es}_2)$. By inductive hypothesis, expressions $\mathtt{es}_1$, $\mathtt{es}_2$ self-stabilise to Ψ_1, Ψ_2 after a certain event. Then, $\mathtt{nbr}(\mathtt{es}_1, \mathtt{es}_2)$ self-stabilise to the corresponding nvalue after one additional full round of execution of each device.
- A converging pattern $\mathtt{es}_c$ = exchange(e, (x) => retsend $\mathtt{f}^{\mathsf{C}}(\mathtt{x}, \mathtt{es}, \overline{\mathtt{e}})$). By inductive hypothesis, es self-stabilises to Ψ after a certain event ϵ_0. Given any index n, let d_n be the maximum distance between $\mathtt{es}_c(\delta')$ and $\Psi(\delta)(\delta')$ for all devices δ' and events ϵ with $\delta = d(\epsilon)$ of the nth round of execution of every device after ϵ_0.
  We prove that d_n is strictly decreasing with n. Since distances are computed on a well-founded set, it follows that they have to became zero for a sufficiently large n, proving self-stabilisation of $\mathtt{es}_c$ to Ψ.
  Consider an event on the nth round of execution of every device after ϵ_0. Thus, supplier events belong to rounds of execution $\geq n-1$, hence their distance with Ψ is at most d_{n-1}. By definition of converging function, it follows that $\mathtt{es}_c$ is strictly closer to Ψ than d_{n-1}, concluding the proof.
- An acyclic pattern $\mathtt{es}_a$ = exchange(e, (x) => retsend $\mathtt{es}_f$(mux($\mathtt{nbrlt}(\mathtt{es}_r), \mathtt{x}, \mathtt{es}_v), \overline{\mathtt{es}}$)). By inductive hypothesis, $\mathtt{es}_f, \mathtt{es}_r, \mathtt{es}_v, \overline{\mathtt{es}}$ self-stabilise to $\Psi_f, \Psi_r, \Psi_v, \overline{\Psi}$ after a certain event ϵ_0.
  Let ϵ be any event after ϵ_0 of the device δ_0 of minimal ranking $\Psi_r(\delta_0)(\delta_0)$ in the network. Since $\Psi_r(\delta_0)(\delta_0)$ is minimal, $\mathtt{nbrlt}(\mathtt{es}_r)$ is false, the mux-expression reduces to $\mathtt{es}_v$ and the whole $\mathtt{es}_a$ to $\mathtt{es}_f(\mathtt{es}_v, \overline{\mathtt{es}})$, which self-stabilises by inductive hypothesis after ϵ_1. Similarly, in any event of the second-minimal ranked device after ϵ_1 the mux-expression is stable, thus ensuring stabilisation after a certain event ϵ_2. The thesis follows by repeating the same reasoning on all devices in order of increasing ranking.
- A minimising pattern $\mathtt{es}_m$ = exchange(e, (x) => retsend $\mathtt{f}^{\mathsf{R}}$(nfold(min, $\mathtt{f}^{\mathsf{MP}}$(x, $\overline{\mathtt{es}}$), es), self(x), $\overline{\mathtt{e}}$)). By inductive hypothesis, $\mathtt{es}, \overline{\mathtt{es}}$ self-stabilise to $\Psi, \overline{\Psi}$ after ϵ_0. Let $P = \overline{\delta}$ be a path in the network graph **G**, and define its *weight* as the result of picking the eventual value $\ell_1 = \Psi(\delta_1)$ of es in the first device δ_1, and repeatedly passing it to subsequent devices through the monotonic progressive function, so that $\ell_{i+1} = \mathtt{f}^{\mathsf{MP}}(\ell_i, \overline{\Psi}(\delta_{i+1})(\delta_i))$. Notice that the weight is well-defined since function $\mathtt{f}^{\mathsf{MP}}$ is required to be stateless. Finally, let Ψ_{out} be such that $\Psi_{\text{out}}(\delta) = \ell_\delta$ is the minimum weight for a path P ending in δ.
  Let $\delta^0, \delta^1, \ldots$ be the list of all devices δ ordered by increasing $\Psi_{\text{out}}(\delta)$. Notice that the path P of minimal weight ℓ_{δ^i} for device i can only pass through nodes such that $\ell_{\delta^j} \leq \ell_{\delta^i}$ (thus s.t. $j < i$). In fact, whenever a path P contains a node j the weight of its prefix until j is at least ℓ_{δ^j}, and any longer prefix has strictly greater weight as $\mathtt{f}^{\mathsf{MP}}$ is progressive.
  We prove by complete induction on i that after a certain event ϵ_{i+1} expression $\mathtt{es}_m$ stabilises to $\Psi_{\text{out}}(\delta^i)$ in device δ^i, and assumes values $\geq \Psi_{\text{out}}(\delta^i)$ in devices δ^j with $j \geq i$. By inductive hypothesis, assume that devices δ^j with $j < i$ are all self-stabilised after ϵ_i. After every device performed an additional round, these values are available in the x variable of exchange. Consider the evaluation of the expression $\mathtt{es}_m$ in a device δ^k with $k \geq i$. Since argument es of nfold is also the weight of the single-node path $P = \delta^k$, it has to be at least $\ell \geq \ell_{\delta^k} \geq \ell_{\delta^i}$. Similarly, the second argument w of nfold for devices δ^j with $j < i$ has to be at least $\mathtt{w}(\delta^j) \geq \ell_{\delta^k} \geq \ell_{\delta^i}$ since it corresponds to weights of (not necessarily minimal) paths P ending in δ^k (obtained by extending a minimal path for a device δ^j with $j < i$ with the additional node δ^k). Finally, values of w for devices δ^j with $j \geq i$ are strictly greater than the minimum value for the whole $\mathtt{es}_m$ expression among all devices δ^j with $j \geq i$, since $\mathtt{f}^{\mathsf{MP}}$ is progressive.
  Thus, as long as the minimum value for the whole expression among non-stable devices is lower than ℓ_{δ^i}, the result of the nfold subexpression is *strictly greater* than this value. The same holds for the overall value, since it is obtained by combining the output of nfold with the previous value for x through the rising function $\mathtt{f}^{\mathsf{R}}$, and a rising function has to be equal to the first argument (the nfold result strictly greater than the minimum), or $\rhd$ than the second. In the latter case, it also needs to be greater or equal to the first argument (again, strictly greater than the minimum) or strictly greater than the second 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 $\rhd$-greater than it.

surpasses ℓ_{δ^i} since $<$ is Noetherian. From that point on, that minimum cannot drop below ℓ_{δ^i}, and the output of `nfold` in δ^i stabilises to ℓ_{δ^i}. In fact, if P is a path of minimum weight for δ^i, then either:

- $P = \delta^i$, so that ℓ_{δ^i} is exactly the third argument of `nfold`, hence also the output of it (since the second argument is greater than ℓ_{δ^i}).
- $P = Q, \delta^i$ where Q ends in δ^j with $j < i$. Since $\mathtt{f}^{\mathsf{MP}}$ is monotonic non-decreasing, the weight of Q', δ^i (where Q' is minimal for δ^j) is not greater than that of P; i.e., $P' = Q', \delta^i$ is also a path of minimum weight. It follows that $\mathtt{w}(\delta^j)$ (where `w` is the second argument of `nfold`) is ℓ_{δ^i}.

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

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, 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[9] 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:

```
1 trait XCLang {
2   def branch[T](cond: NValue[Boolean])(th: =>
        NValue[T])(el: => NValue[T]): NValue[T]
3   def exchange[T](init: NValue[T])(f: NValue[T]
        => (NValue[T],NValue[T])): NValue[T]
4 }
```

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

```
1 class NValue[T](val defaultMessage: T, val
      customMessages: Map[ID,T] = Map.empty) {
2   def fold[V>:T](init: V)(f: (V,V)=>V): NValue[V]
        = // ...
3   def map2[R,S](other: NValue[R])(m: (T,R)=>S):
        NValue[S] = // ...
4   // more built-ins ... (cf. Fig. 2)
5 }
```

[9] 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).

We leverage Scala implicit conversions and extension methods (Oliveira et al., 2010), imported by mixing in `XCLib`, to automatically convert values of type `T` to `NValues` of `T`s and, e.g., to extend `NValues` of `Numerics` to accept operators like `+` (to combine nvalues pointwise). An abstract class `XCProgram[T]` requires programmers to override the method `main:T`. Moreover, it exposes methods `sense` and `senseNeighbour` to subclasses for retrieving local and neighbouring values from the execution environment. For instance, the gradient program (Example 3) can be encoded as follows.

```
1 object gradient extends XCProgram[Double] with
      XCLib {
2   def main =
3     exchange(Double.PositiveInfinity)(n =>
4       mux(sense[Boolean]("source")){ 0.0 }{
5         (n + senseNeighbour("distance")).fold(
              Double.PositiveInfinity)(Math.min)
6       })
7 }
```

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:

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

In this implementation, message communication occurs only *before* (Line 3) and *after* (Line 7) the firing (Line 5) to ensure that the `exchange` within the round are all executed atomically w.r.t. the messages that are received and sent by the device (Section 2.1). 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: `getData()` to obtain values from the local environment, `getMessages()` to retrieve messages from neighbours (e.g., a peer node may keep them in a buffer, a cloud platform may use an in-memory database service, a simulator may use an ad-hoc map-like data structure), `process()` to drive actuations (e.g., locally on a node, or through a command on a cloud back-end), and `propagate()` to send exported data to neighbours (e.g., through a direct message to the neighbour, or through a write on shared state in simulations or cloud).

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

Fig. 13. Implementation of `gossipEver` in C++/XC.

6.2. C++ DSL

We implemented XC as a C++ DSL,[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 summarising 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).

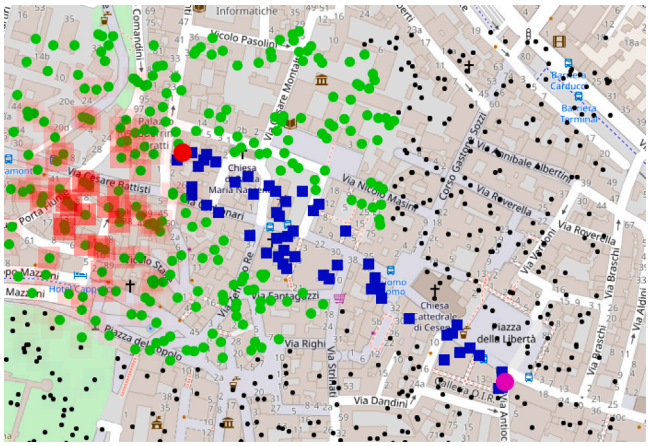
No external dependencies are needed for those back-ends.

7. Evaluation

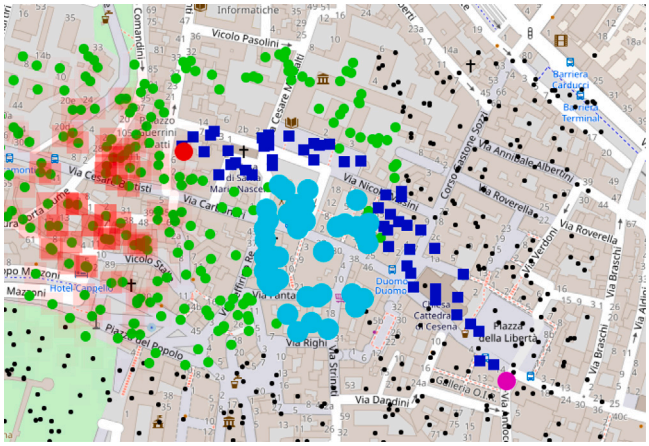
In this section, we evaluate XC.[11] The goal is to show that *(RQ1)* the decentralised execution of the XC program on each device results in the desired *collective* behaviour and that *(RQ2)* the overall behaviour can be expressed by composing functions of collective behaviour that correctly combine 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/scafi/artifact-2021-ecoop-smartc and permanently as an archived artefact on Zenodo (Casadei, 2022a). This artefact is also described by the artefact paper (Audrito et al., 2022b).



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



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

Fig. 14. Two snapshots of the SmartC case study.

7.1. SmartC case study

We consider a simulation of the SmartC scenario described in Example 11, and we implement it both in the Scala and C++ DSLs (the results in this section refer to the Scala implementation). We believe that other application domains, such as WSNs and WSNs, would not pose fundamentally different challenges compared to the considered scenario: WSNs focus on information flows, which is part of the case study, and CPSs emphasise on actuation, which could be a simple variant of the scenario, e.g., where agents move according to the gathered reports. In the simulation setup, 600 devices each running the XC program communicate with every neighbour currently in a 50-metre range once per second. We consider a single detector and a single operations centre. The simulator enables the collection of data exported at the individual nodes (i.e., the program Example 11 is extended with simulation-specific code). We measure, every second, the actual (instantaneous) mean warning in the inspection area (using an oracle, namely a process that can inspect the simulated system at any instant) and the mean warning measured by the operations centre. We consider the average result over 30 simulations varying the actual displacement of devices and scheduling offsets. We inject a blackout event that disconnects a set of devices from the system, hence disrupting the channel. Fig. 14 shows two snapshots of the simulation with devices (black dots), detector (red dot), sensors within the area inspected by the detector (green dots), operations centre (magenta dot), and inoperable devices (cyan dots). Semi-transparent red squares denote the warning

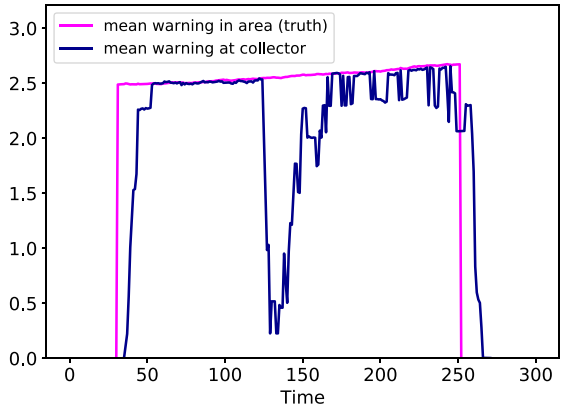


Fig. 15. Execution of SmartC.

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 jags in the second blue wave are due to perturbations (exacerbated by the obstacle) that temporarily destroy the channel in a few simulations, while delays depend on the firing frequency and communication hops (from inspection area edges to detector to operations centre).

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, S_{PS}, *(ii)* a pub-sub XC-like solution S_{PSXC} with a design inspired by XC, and *(iii)* an XC solution S_{XC}. This allows us to draw some interesting indications on the compactness that programming in XC can provide.

The core programs are 82, 28, 22 LoC long. In S_{PS}, *the logic spreads over multiple subscription handlers*, while in S_{PSXC} and S_{XC} the core logic is neatly separated. The S_{PS} version uses 4 handlers (and crucially, any additional field would need a further handler), 2 `send`s, and 4 `publish`es, while S_{PSXC} uses 2, 1, and 3 resp. Also, S_{PS} keeps 6 state variables for the input context of a device–S_{PSXC} only 3. W.r.t. S_{XC}, S_{PSXC} has a *coding overhead due to the topics management* and to the more *brittle handling of neighbour data*, of about 27% more LoC, 73% more words, and 35% more method calls. Finally, the main limitation of S_{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), formal properties of programs and computations (Viroli et al., 2018; Beal et al., 2017; Audrito et al., 2018a), programming language implementations of formal calculi as DSLs (Casadei et al., 2022b, 2021; Audrito, 2020), simulators (Pianini et al., 2013; Audrito et al., 2022e), algorithms and patterns (Beal, 2009; Audrito et al., 2017b,a; 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, 2022a), 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).

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

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, Viroli 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 Viroli 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 predicated 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 Viroli 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)* (Viroli et al., 2018; Audrito et al., 2020), which is the reference model for computational fields (Viroli 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 $\mathrm{v} = \ell[\overline{\delta} \mapsto \overline{\ell}]$. In particular, *local values* are equivalent to nvalues $\ell[]$ without custom messages, and *neighbouring values* are equivalent to nvalues with any valid default message. This unification allows a simpler type system and, crucially, differentiated messages to neighbours.

By interpreting FC values as nvalues, all FC message-exchanging constructs (`nbr`, `rep` Viroli et al., 2018 and `share` Audrito et al., 2020) can be modelled within XC: `nbr` is the same defined function introduced in Section 2.2, just restricted to operate on local values only; `share` corresponds to an `exchange` with `retsend` restricted to operate on local values only; and `rep`$(e_1)\{(x) \Rightarrow e_2\}$ can be translated to `exchange`$(e_1,(x) \Rightarrow$ `retsend` $e_2[x :=$ `self`$(x)])$. Notice that the converse translation is not possible, as `nbr`, `rep` or `share` expressions with arguments of neighbouring type have no defined behaviour in FC. Thus, `nbr` and `exchange` in XC are strictly more expressive than their corresponding FC counterparts `nbr` and `share`: they can be used with expressions producing nvalues with custom messages to model differentiated messages.

The properties for subsets of the *field calculus* (FC), as surveyed in Viroli et al. (2019), include 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), 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 `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; Viroli 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 Viroli 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" (2020HCWWLP), 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

Abd Alrahman, Y., De Nicola, R., Loreti, M., 2020. Programming interactions in collective adaptive systems by relying on attribute-based communication. Sci. Comput. Program. 192, http://dx.doi.org/10.1016/j.scico.2020.102428.

Abowd, G.D., 2016. Beyond weiser: From ubiquitous to collective computing. Computer 49 (1), 17–23. http://dx.doi.org/10.1109/MC.2016.22.

Aguzzi, G., Casadei, R., Viroli, M., 2023. MacroSwarm: A field-based compositional framework for swarm programming. In: Coordination Models and Languages - 25th International Conference, COORDINATION'23, Proceedings. In: LNCS, vol. 13908, Springer, pp. 31–51. http://dx.doi.org/10.1007/978-3-031-35361-1_2.

Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W., 1995. Causal memory: Definitions, implementation, and programming. Distrib. Comput. 9 (1), 37–49. http://dx.doi.org/10.1007/BF01784241.

Ancona, D., Dagnino, F., Zucca, E., 2017a. Generalizing inference systems by coaxioms. In: Programming Languages and Systems - 26th European Symposium on Programming, ESOP'17, Proceedings. In: LNCS, vol. 10201, Springer, pp. 29–55. http://dx.doi.org/10.1007/978-3-662-54434-1_2.

Ancona, D., Dagnino, F., Zucca, E., 2017b. Reasoning on divergent computations with coaxioms. Proc. ACM Program. Lang. 1 (OOPSLA), 81:1–81:26. http://dx.doi.org/10.1145/3133905.

Ancona, D., Dagnino, F., Zucca, E., 2018. Modeling infinite behaviour by corules. In: 32nd European Conference on Object-Oriented Programming. ECOOP'18, In: LIPIcs, vol. 109, Schloss Dagstuhl, pp. 21:1–21:31. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2018.21.

Artho, C., Havelund, K., Kumar, R., Yamagata, Y., 2015. Domain-specific languages with Scala. In: ICFEM. In: Lecture Notes in Computer Science, vol. 9407, Springer, pp. 1–16. http://dx.doi.org/10.1007/978-3-319-25423-4_1.

Atzori, L., Iera, A., Morabito, G., 2010. The Internet of Things: A survey. Comput. Netw. 54 (15), 2787–2805. http://dx.doi.org/10.1016/j.comnet.2010.05.010.

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

Audrito, G., Beal, J., Damiani, F., Pianini, D., Viroli, M., 2020. Field-based coordination with the share operator. Log. Methods Comput. Sci. 16 (4), http://dx.doi.org/10.23638/LMCS-16(4:1)2020.

Audrito, G., Beal, J., Damiani, F., Viroli, M., 2018a. Space-time universality of field calculus. In: Coordination Models and Languages. In: Lecture Notes in Computer Science, vol. 10852, Springer, pp. 1–20. http://dx.doi.org/10.1007/978-3-319-92408-3_1.

Audrito, G., Casadei, R., Damiani, F., Pianini, D., Viroli, M., 2021a. Optimal resilient distributed data collection in mobile edge environments. Comput. Electr. Eng. http://dx.doi.org/10.1016/j.compeleceng.2021.107580.

Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M., 2022a. Functional programming for distributed systems with XC. In: 36th European Conference on Object-Oriented Programming, ECOOP 2022. In: LIPIcs, vol. 222, Schloss Dagstuhl, pp. 20:1–20:28. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2022.20.

Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M., 2022b. Functional programming for distributed systems with XC (artifact). Dagstuhl Artifacts Ser. 8 (2), 08:1–08:4. http://dx.doi.org/10.4230/DARTS.8.2.8.

Audrito, G., Casadei, R., Damiani, F., Stolz, V., Viroli, M., 2021b. Adaptive distributed monitors of spatial properties for cyber-physical systems. J. Syst. Softw. 175, 110908. http://dx.doi.org/10.1016/j.jss.2021.110908.

Audrito, G., Casadei, R., Damiani, F., Viroli, M., 2017a. Compositional blocks for optimal self-healing gradients. In: Self-Adaptive and Self-Organizing Systems (SASO), 2017. IEEE, IEEE Computer Society, pp. 91–100. http://dx.doi.org/10.1109/SASO.2017.18.

Audrito, G., Casadei, R., Damiani, F., Viroli, M., 2023a. Computation against a neighbour: Addressing large-scale distribution and adaptivity with functional programming and Scala. Log. Methods Comput. Sci. 19 (1), http://dx.doi.org/10.46298/lmcs-19(1:6)2023.

Audrito, G., Damiani, F., Stolz, V., Torta, G., Viroli, M., 2022c. Distributed runtime verification by past-CTL and the field calculus. J. Syst. Softw. 187, 111251. http://dx.doi.org/10.1016/j.jss.2022.111251.

Audrito, G., Damiani, F., Torta, G., 2022d. Bringing aggregate programming towards the cloud. In: 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA 2022, Proceedings, Part III. In: LNCS, vol. 13703, Springer, pp. 301–317. http://dx.doi.org/10.1007/978-3-031-19759-8_19.

Audrito, G., Damiani, F., Viroli, M., 2017b. Optimally-self-healing distributed gradient structures through bounded information speed. In: COORDINATION 2017. In: LNCS, vol. 10319, Springer, pp. 59–77. http://dx.doi.org/10.1007/978-3-319-59746-1_4, Best paper at COORDINATION'17.

Audrito, G., Damiani, F., Viroli, M., Bini, E., 2018b. Distributed real-time shortest-paths computations with the field calculus. In: 2018 IEEE Real-Time Systems Symposium. RTSS, IEEE Computer Society, pp. 23–34. http://dx.doi.org/10.1109/RTSS.2018.00013.

Audrito, G., Rapetta, L., Torta, G., 2022e. Extensible 3D simulation of aggregated systems with FCPP. In: Coordination Models and Languages - 24th International Conference, COORDINATION 2022 Proceedings. In: LNCS, vol. 13271, Springer, pp. 55–71. http://dx.doi.org/10.1007/978-3-031-08143-9_4.

Audrito, G., Terraneo, F., Fornaciari, W., 2023b. FCPP+Miosix: Scaling aggregate programming to embedded systems. IEEE Trans. Parallel Distrib. Syst. 34 (3), 869–880. http://dx.doi.org/10.1109/TPDS.2022.3232633.

Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J., 2019. A higher-order calculus of computational fields. ACM Trans. Comput. Log. 20 (1), 5:1–5:55. http://dx.doi.org/10.1145/3285956.

Beal, J., 2009. Flexible self-healing gradients. In: ACM Symposium on Applied Computing, Proceedings. SAC '09, ACM, pp. 1197–1201. http://dx.doi.org/10.1145/1529282.1529550.

Beal, J., 2016. Trading accuracy for speed in approximate consensus. Knowl. Eng. Rev. 31 (4), 325–342. http://dx.doi.org/10.1017/S0269888916000175.

Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N., 2013. Organizing the aggregate: Languages for spatial computing. In: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global, pp. 436–501. http://dx.doi.org/10.4018/978-1-4666-2092-6.ch016, Ch. 16.

Beal, J., Pianini, D., Viroli, M., 2015. Aggregate programming for the Internet of Things. IEEE Comput. 48 (9), http://dx.doi.org/10.1109/MC.2015.261.

Beal, J., Viroli, M., Pianini, D., Damiani, F., 2017. Self-adaptation to device distribution in the Internet of Things. ACM Trans. Auton. Adapt. Syst. 12 (3), 12:1–12:29. http://dx.doi.org/10.1145/3105758.

Brambilla, M., Ferrante, E., Birattari, M., Dorigo, M., 2013. Swarm robotics: a review from the swarm engineering perspective. Swarm Intell. 7 (1), 1–41. http://dx.doi.org/10.1007/s11721-012-0075-2.

Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F., 2013. DEECO: an ensemble-based component system. In: Symposium on Component Based Software Engineering. CBSE, ACM, pp. 81–90. http://dx.doi.org/10.1145/2465449.2465462.

Casadei, R., 2022a. scafi/artifact-2021-ecoop-smartc: v1.2. http://dx.doi.org/10.5281/ZENODO.6538822, URL https://zenodo.org/record/6538822.

Casadei, R., 2022b. scafi/artifact-2021-ecoop-xc: v1.2. http://dx.doi.org/10.5281/ZENODO.6538810, URL https://zenodo.org/record/6538810.

Casadei, R., 2023. Macroprogramming: Concepts, state of the art, and opportunities of macroscopic behaviour modelling. ACM Comput. Surv. http://dx.doi.org/10.1145/3579353.

Casadei, R., Fortino, G., Pianini, D., Placuzzi, A., Savaglio, C., Viroli, M., 2022a. A methodology and simulation-based toolchain for estimating deployment performance of smart collective services at the edge. IEEE Internet Things J. 9 (20), 20136–20148. http://dx.doi.org/10.1109/JIOT.2022.3172470.

Casadei, R., Pianini, D., Placuzzi, A., Viroli, M., Weyns, D., 2020. Pulverization in cyber-physical systems: Engineering the self-organizing logic separated from deployment. Future Internet 12 (11), 203. http://dx.doi.org/10.3390/fi12110203.

Casadei, R., Viroli, M., Aguzzi, G., Pianini, D., 2022b. ScaFi: A scala DSL and toolkit for aggregate programming. SoftwareX 20, 101248. http://dx.doi.org/10.1016/j.softx.2022.101248.

Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F., 2019. Aggregate processes in field calculus. In: Coordination Models and Languages - 21st International Conference, COORDINATION 2019 Proceedings. In: LNCS, vol. 11533, Springer, pp. 200–217. http://dx.doi.org/10.1007/978-3-030-22397-7_12.

Casadei, R., Viroli, M., Audrito, G., Pianini, D., Damiani, F., 2021. Engineering collective intelligence at the edge with aggregate processes. Eng. Appl. Artif. Intell. 97, 104081. http://dx.doi.org/10.1016/j.engappai.2020.104081.

Cousot, P., Cousot, R., 1992. Inductive definitions, semantics and abstract interpretation. In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, pp. 83–94. http://dx.doi.org/10.1145/143165.143184.

Dagnino, F., 2019. Coaxioms: flexible coinductive definitions by inference systems. Log. Methods Comput. Sci. 15 (1), http://dx.doi.org/10.23638/LMCS-15(1:26)2019.

Dagnino, F., 2022. A meta-theory for big-step semantics. ACM Trans. Comput. Log. 23 (3), 20:1–20:50. http://dx.doi.org/10.1145/3522729.

Damas, L., Milner, R., 1982. Principal type-schemes for functional programs. In: Symposium on Principles of Programming Languages. POPL '82, ACM, pp. 207–212. http://dx.doi.org/10.1145/582153.582176.

Dasgupta, S., Beal, J., 2016. A Lyapunov analysis for the robust stability of an adaptive Bellman-Ford algorithm. In: Decision and Control (CDC), 2016 IEEE 55th Conference on. IEEE, pp. 7282–7287. http://dx.doi.org/10.1109/CDC.2016.7799393.

De Nicola, R., Loreti, M., Pugliese, R., Tiezzi, F., 2014. A formal approach to autonomic systems programming: The SCEL language. ACM Trans. Auton. Adapt. Syst. 9 (2), 7:1–7:29. http://dx.doi.org/10.1145/2619998.

DeHon, A., Giavitto, J., Gruau, F. (Eds.), 2007. Computing Media and Languages for Space-Oriented Computation. In: Dagstuhl Seminar Proceedings, vol. 06361, Schloss Dagstuhl, URL http://drops.dagstuhl.de/portals/06361.

Doeraene, S., 2018. Cross-platform language design in Scala.js (keynote). In: SCALA@ICFP. ACM, p. 1. http://dx.doi.org/10.1145/3241653.3266230.

Ferscha, A., 2015. Collective adaptive systems. In: UbiComp/ISWC Adjunct. ACM, pp. 893–895. http://dx.doi.org/10.1145/2800835.2809508.

Ghosh, D., Sheehy, J., Thorup, K.K., Vinoski, S., 2012. Programming language impact on the development of distributed systems. J. Internet Serv. Appl. 3 (1), 23–30. http://dx.doi.org/10.1007/s13174-011-0042-y.

Gummadi, R., Gnawali, O., Govindan, R., 2005. Macro-programming wireless sensor networks using Kairos. In: Distributed Computing in Sensor Systems. DCOSS, pp. 126–140. http://dx.doi.org/10.1007/11502593_12.

Igarashi, A., Pierce, B.C., Wadler, P., 2001. Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23 (3), http://dx.doi.org/10.1145/503502.503505.

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21 (7), 558–565. http://dx.doi.org/10.1145/359545.359563.

Leroy, X., Grall, H., 2009. Coinductive big-step operational semantics. Inform. and Comput. 207 (2), 284–304. http://dx.doi.org/10.1016/J.IC.2007.12.004.

Lluch-Lafuente, A., Loreti, M., Montanari, U., 2017. Asynchronous distributed execution of fixpoint-based computational fields. Log. Methods Comput. Sci. 13 (1), http://dx.doi.org/10.23638/LMCS-13(1:13)2017.

Malone, T.W., Crowston, K., 1994. The interdisciplinary study of coordination. ACM Comput. Surv. 26 (1), 87–119. http://dx.doi.org/10.1145/174666.174668.

Mamei, M., Zambonelli, F., 2004. Programming pervasive and mobile computing applications with the TOTA middleware. In: Pervasive Computing and Communications, 2004. IEEE, pp. 263–273. http://dx.doi.org/10.1109/PERCOM.2004.1276864.

Mamei, M., Zambonelli, F., 2006. Field-Based Coordination for Pervasive Multiagent Systems. In: Springer Series on Agent Technology, Springer, http://dx.doi.org/10.1007/3-540-27969-5.

Masolo, C., Vieu, L., Ferrario, R., Borgo, S., Porello, D., 2020. Pluralities, collectives, and composites. In: Formal Ontology in Information Systems, Proceedings, FOIS 2020. In: Frontiers in Artificial Intelligence and Applications, vol. 330, IOS Press, pp. 186–200. http://dx.doi.org/10.3233/FAIA200671.

Mills, K.L., 2007. A brief survey of self-organization in wireless sensor networks. Wirel. Commun. Mob. Comput. 7 (7), 823–834. http://dx.doi.org/10.1002/wcm.499.

Morin, C., Puaut, I., 1997. A survey of recoverable distributed shared virtual memory systems. IEEE Trans. Parallel Distrib. Syst. 8 (9), 959–969. http://dx.doi.org/10.1109/71.615441.

Mottola, L., Picco, G.P., 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Comput. Surv. 43 (3), 19:1–19:51. http://dx.doi.org/10.1145/1922649.1922656.

Murphy, A.L., Picco, G.P., Roman, G., 2006. LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15 (3), 279–328. http://dx.doi.org/10.1145/1151695.1151698.

Mushtaq, H., Al-Ars, Z., Bertels, K., 2011. Survey of fault tolerance techniques for shared memory multicore/multiprocessor systems. In: International Design and Test Workshop. IDT, IEEE, pp. 12–17. http://dx.doi.org/10.1109/IDT.2011.6123094.

Newton, R., Welsh, M., 2004. Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks. pp. 78–87. http://dx.doi.org/10.1145/1052199.1052213.

Nicola, R.D., Jähnichen, S., Wirsing, M., 2020. Rigorous engineering of collective adaptive systems: special section. Int. J. Softw. Tools Technol. Transf. 22 (4), 389–397. http://dx.doi.org/10.1007/s10009-020-00565-0.

Oliveira, B.C.d.S., Moors, A., Odersky, M., 2010. Type classes as objects and implicits. In: SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA, ACM, pp. 341–360. http://dx.doi.org/10.1145/1869459.1869489.

Pianini, D., Casadei, R., Viroli, M., 2022. Self-stabilising priority-based multi-leader election and network partitioning. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems. ACSOS 2022, IEEE, pp. 81–90. http://dx.doi.org/10.1109/ACSOS55765.2022.00026.

Pianini, D., Casadei, R., Viroli, M., Mariani, S., Zambonelli, F., 2021a. Time-fluid field-based coordination through programmable distributed schedulers. Log. Methods Comput. Sci. 17 (4), http://dx.doi.org/10.46298/lmcs-17(4:13)2021.

Pianini, D., Casadei, R., Viroli, M., Natali, A., 2021b. Partitioned integration and coordination via the self-organising coordination regions pattern. Future Gener. Comput. Syst. 114, 44–68. http://dx.doi.org/10.1016/j.future.2020.07.032.

Pianini, D., Montagna, S., Viroli, M., 2013. Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simul. 7 (3), 202–215. http://dx.doi.org/10.1057/jos.2012.27.

Pierce, B.C., 2002. Types and Programming Languages. MIT Press.

Sene Júnior, I.G., Santana, T.S., Bulcão-Neto, R.F., Porter, B., 2022. The state of the art of macroprogramming in IoT: An update. J. Internet Serv. Appl. 13 (1), 54–65. http://dx.doi.org/10.5753/jisa.2022.2372.

Testa, L., Audrito, G., Damiani, F., Torta, G., 2022. Aggregate processes as distributed adaptive services for the Industrial Internet of Things. Pervasive Mob. Comput. 85, 101658. http://dx.doi.org/10.1016/j.pmcj.2022.101658.

Tumer, K., Wolpert, D. (Eds.), 2004. Collectives and the Design of Complex Systems. Springer, New York, NY.

Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D., 2018. Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. 28 (2), 16:1–16:28. http://dx.doi.org/10.1145/3177774.

Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D., 2019. From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. 109, http://dx.doi.org/10.1016/j.jlamp.2019.100486.

Weisenburger, P., Köhler, M., Salvaneschi, G., 2018. Distributed system development with ScalaLoci. Proc. ACM Program. Lang. 2 (OOPSLA), 129:1–129:30. http://dx.doi.org/10.1145/3276499.

Weisenburger, P., Wirth, J., Salvaneschi, G., 2020. A survey of multitier programming. ACM Comput. Surv. 53 (4), 81:1–81:35. http://dx.doi.org/10.1145/3397495.

Zhang, K., Yang, Z., Basar, T., 2019. Multi-agent reinforcement learning: A selective overview of theories and algorithms. CoRR abs/1911.10635. arXiv:1911.10635.

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.

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).

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.