Distributed Runtime Verification in Proximity-based Networks: A Tutorial on the Aggregate Programming Approach

Giorgio Audrito[1 [0000−0002−2319−0375]], Ferruccio Damiani[1 [0000−0001−8109−1706]], Giordano Scarso[1 [0009−0009−2114−7435]], Volker Stolz[2 [0000−0002−1031−6936]], and Gianluca Torta[1 [0000−0002−4276−7213]]

[1] Department of Computer Science, University of Turin, Torino, Italy
{giorgio.audrito,ferruccio.damiani,giordano.scarso,gianluca.torta}@unito.it
[2] Department of Computer science, Electrical engineering and Mathematical sciences, Western Norway University of Applied Sciences, Bergen, Norway
Volker.Stolz@hvl.no

Abstract. Distributed runtime verification (DRV) addresses the problem of checking the correctness of distributed systems during execution, coping with partial knowledge, dynamic topologies, and the absence of global time. These challenges are particularly prominent in proximity-based networks, such as those arising in IoT and Far Edge computing scenarios, where large numbers of devices interact through local communication. This tutorial presents an approach to DRV based on Aggregate Programming (AP), a paradigm for designing distributed collective systems via high-level abstractions over computational fields. We show how temporal and spatial properties (expressed in past-CTL and SLCS, respectively) can be systematically compiled into aggregate monitors grounded in the eXchange Calculus and executed using the FCPP C++ framework and simulator for AP. The tutorial combines conceptual foundations with practical guidance: participants learn how to specify spatio-temporal properties, generate corresponding monitors, and execute them in a 3D simulation environment. Examples are drawn from ongoing industrial collaborations and research projects, which we use to illustrate realistic monitoring scenarios and motivate open challenges for AP-based DRV.

Keywords: Aggregate Programming · Distributed Runtime Verification · Self-Stabilisation.

1 Introduction

What is Distributed Runtime Verification? *Runtime verification (RV)* is a program analysis paradigm that focuses on observing a system during execution in order to check whether its behaviour conforms to a given specification, using monitors generated from formal specifications [26]. *Distributed runtime verification (DRV)* extends this paradigm to distributed systems, encompassing

both the monitoring of distributed systems and the use of distributed infrastructures to perform monitoring. As a verification technique, runtime verification emphasizes the systematic generation of monitors from formal specifications, enabling properties to be stated precisely and providing formal guarantees about the correctness of the monitoring outcomes. Distribution, however, introduces additional challenges for verification, as it requires addressing issues such as synchronization, communication faults, and the absence of a unique global time [23].

What are Proximity-based Networks? *Proximity-based networks* are (possibly dense and large) networks of devices (possibly mobile, ephemeral, capable of computation, communication and interaction with the environment and people) where:

- the topology is dynamic, induced by the proximity (physical or logical) of the devices; and
- each device can receive/send messages to/from devices in the neighbourhood.

These arise naturally in relation to Far Edge Computing, which brings together all locally installed IT systems capable of operating independently of the Cloud and can form the basis for decentralised IT architectures. Far Edge Computing represents an opportunity for the EU's industrial ecosystem, which is strong in professional Internet of Things (IoT) applications [30].

What is AP? *Aggregate Programming (AP)* [15] is an established paradigm for engineering distributed systems that has been actively developed for over a decade [38]. Its execution model (typically implemented within a runtime) assumes that each device repeatedly executes asynchronous sense–compute–interact rounds. The paradigm is formally grounded in AP calculi, such as the *Field Calculus (FC)* [14] and its recent extension *eXchange Calculus (XC)* [5]. These are functional core languages designed to express computations over distributed data structures, called *computational fields* [27], that map sets of devices to values, representing collective inputs and outputs. This formal foundation enables rigorous reasoning about aggregate programs, including the derivation of guarantees and proofs (for example, establishing that a collective computation is self-stabilising, meaning that it eventually converges to the correct result once its inputs have stabilized [37]). Aggregate Programming languages are realized through standalone or embedded domain-specific languages (DSLs) [38], which are often accompanied by libraries of reusable functions capturing common patterns of collective behaviour. Using these languages and libraries, programmers write a single, global program that specifies the overall behaviour of the collective as a composition of functions. This program can then be deployed over a network of devices, either simulated or, through middleware support, made of physical systems [17]. By separating the logical specification of the system from its physical deployment, AP also supports partitioning applications into deployable components, enabling computation to be offloaded across the (Far) Edge–Cloud Continuum [22]. Currently, the main way to analyse AP programs is by simulation. The AP-based DRV techniques discussed here, which can of course be applied in particular to AP systems, can add a valuable tool for (online) checking of formal properties.

Practical Support for AP. The main practical support for AP is given by: the Scala internal DSL, library and toolchain *ScaFi* [33] (see also [18,19,16,7]); the C++ internal DSL, library and toolchain *FCPP* [36] (see also [2,11,12,13]); and the Kotlin internal DSL, compiler plugin, library and toolchain *Collektive* [34] (see also [21]). FCPP allows programmers to embed XC programs in C++ applications, and deploy them on networks of physical devices, as well as a simulator. It is used in a number of industrial case studies currently developed in collaboration with the companies Concept Reply (https://www.reply.com/concept-reply/en), Synesthesia (https://synesthesia.it), Eurix (https://www.eurixgroup.com), ST Microelectronics (https://www.st.com/content/st_com/en.html), and Torino Airport (https://www.aeroportoditorino.it/en)[3] – preliminary results related to these activities have been published [32,8,28].

This Tutorial, its Accompanying Material, and Further Reading. This tutorial caters to all who want to learn about DRV in proximity-based networks using AP: newcomers to the field as well as experienced researchers in formal methods outside of distributed programming and RV. It offers an overview of DRV, of the past-CTL (an extension of past-LTL [24]) temporal logic and the SLCS [20] spatial logic for specifying monitors over proximity-based networks, on AP, on how to generate AP monitors from past-CTL and SLCS formulas, and on how to run them in the FCPP simulator. Participants of the live conference tutorial gain hands-on experience with the FCPP framework. By the end of the tutorial, they will be able to actively use FCPP for specifying (simple) monitors, and running them on the FCPP simulator. Videos, slides, and all examples of the conference tutorial, as well as the FCPP framework itself, are available for download at https://fcpp.github.io/quickstart/tutorial-fm-2026.html. For further reading, two journal papers present the AP approach for past-CTL monitors [9] and SLCS monitors [6], a journal paper presents the formal foundation of XC [5], and the FCPP website (https://fcpp.github.io) provides an up-to-date list of papers related to DRV by AP.

Paper Organization. Section 2 presents DRV from a theoretical perspective. Section 3 introduces aggregate programming with the XC language and FCPP. Finally, Section 4 describes an automatic synthesis of AP monitors for spatio-temporal formulas, and presents experiments of DRV in FCPP. Section 5 provides a recollection of AP case studies and how DRV could be applied to them.

2 The Theory of DRV in Proximity-based Networks

Distributed runtime verification is a sub-field of the more general field of runtime verification, where specifications over events as atomic propositions are evaluated on traces or streams [26]. Popular specification languages include variations on

[3] See, e.g., the RoboAPP [35] and the RoboNG [31] cascade funding projects of the Spoke 1 "Sustainable mobility and aerospace" of the PNRR NODES (https://ecs-nodes.eu/en); and the CN AgriTech (https://agritechcenter.it).

the Linear Time Logic LTL, and regular expressions. Events may be generated through state changes or execution flow, such as method calls.

In distributed runtime verification [23], agents (representing the part of the system to verify at each device) are generally considered *remote* to each other: as constituents of the whole system, they are assumed to execute independently and occasionally synchronise or communicate with each other via the underlying communication platform. A *local trace* of *events* corresponds to a sequence of sets of values for observables, as defined through the sensors of an agent, or derived values from those. Since agents may appear or disappear over time from the overall system, traces from different agents are not aligned in time in the sense that for a particular index/position in each trace, these events did not necessarily happen at the same time. Accordingly, logic formulas cannot state properties over single traces (except in the degenerate case of specifying a property only on the history of a single agent). Therefore, we will first introduce a suitable computational model through augmented event structures, before we define our logics for the temporal and the spatial fragments which will allow us to draw conclusions from the shared histories of agents, that evolve intermittently only locally on each device in-between synchronizations with peers within communication range. The partially or fully evaluated spatio-temporal properties as seen from a particular agent can hence be seen as a compressed history wrt. to the values of these properties.

2.1 The Execution Model

In earlier work, we have identified Lamport's notion of *event structures* as a suitable view on the overall system [25,6]. This representation has been used successfully already before, to model the partial order relation between events arising from model causality (communication across agents, or agent internal computation steps, where agents are identified by device identifiers) [40,14]. We paraphrase the necessary background from [6]:

Definition 1 (Augmented Event Structure). *An* augmented event structure $\mathbb{E} = \langle E, \rightsquigarrow, < \rangle$ *is a finite or countably infinite set of* events E *together with a neighbouring relation* $\rightsquigarrow \subseteq E \times E$ *and a causality relation* $< \subseteq E \times E$*, such that the transitive closure of* $\rightsquigarrow$ *forms the irreflexive partial order* $<$*, and the set* $X_\epsilon = \{\epsilon' \in E \mid \epsilon' < \epsilon\} \cup \{\epsilon' \in E \mid \epsilon \rightsquigarrow \epsilon'\}$ *is finite for all* ϵ *(i.e.,* $<$ *and* $\rightsquigarrow$ *are locally finite). Thus, we say that* ϵ' *is a neighbour of* ϵ *iff* $\epsilon' \rightsquigarrow \epsilon$*, and that* $\mathcal{N}(\epsilon) = \{\epsilon' \in E \mid \epsilon' \rightsquigarrow \epsilon\}$ *is the set of neighbours of* ϵ*.*

Figure 1 depicts a sample augmented event structure, showing the $\rightsquigarrow$ relation, and how the $<$-relation partitions events into "causal past" (red), "causal future" (cyan), and non-ordered "concurrent" (black) subspaces with respect to any given event (in Figure 1, colours reflect causality with respect to the doubly-circled event in magenta). In principle, an execution at ϵ can depend on information from any event in its past and its results can influence any event in its future. Causality is uniquely induced by neighbouring (the $\rightsquigarrow$ relation),

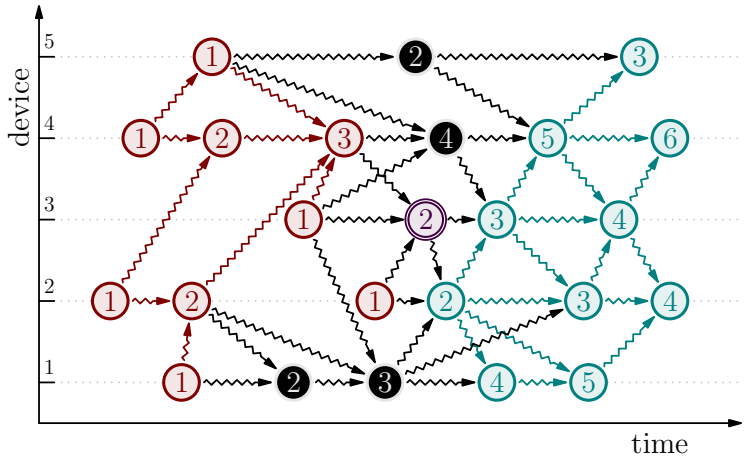

Fig. 1. Example of an augmented event structure, comprising events (circles), neighbour relations (arrows), devices (ordinate axis). Colours indicate causal structure with respect to the doubly-circled event (magenta), splitting events into past (red), future (cyan) and concurrent (non-ordered, in black). The numbers within events represent a space-time value associated with that event, namely a counter value which is reset if the device crashes, see device 2. Note that the set of neighbouring events $\mathcal{N}(\epsilon)$ of the doubly-circled event has three elements: event 1 at the same device (its previous firing), event 3 at device 4, and event 1 at device 2. Figure taken from [3].

dictating when an event can *directly* influence (by message-passing) another. Intuitively, every $\rightsquigarrow$ relation correspond to the send and receive of a message: in order for $\epsilon_1 \rightsquigarrow \epsilon_2$ to hold, execution of event ϵ_1 on a device δ_1 must result in a message which reaches a device δ_2 before its execution of ϵ_2. The name *neighbouring* reflects that message exchanges happen on devices that are close to each other (in some physical or logical sense).

Any sequence of computation events and message exchanges between them can be represented as an augmented event structure, however, not all event structures are physically realisable by a distributed system following the firing model described at the beginning of this section. We refer the reader to [6] for a detailed definition of the subset of *realisable event structures* which additionally satisfy coherence constraints on linearity and uniqueness (a single device only has a single history), as well as impersistence ("reboots" lose history) and computation immediacy (acyclic event relation).

2.2 Temporal Logic: past-CTL

Past-CTL is a variant of the more standard future-time Computation Tree Logic (CTL). Here, operators look back in time, and we have chosen corresponding names dual to their well-known future counterparts. Note that past-CTL is suitable for monitoring, since it refers only to the past and each formula therefore

has a definite truth value at any given time. In contrast, future-time logics such as CTL are more suitable for tasks such as model checking.

Atomic propositions get a (possibly different) truth value for each particular event. Path quantifiers distinguish if properties hold on *all* or *some* paths starting from some initial event of a device and ending in the current event. Unquantified formulas are hence linear properties on the past on a single device.

$\mathrm{Y}\,\phi$	*yesterday*, ϕ held in the previous event on the considered path (on the same device for Y, on every neighbour device for AY, on some device for EY)
$\psi\,\mathrm{S}\,\phi$	*since*, the second argument held in some past event in the considered path, and its first argument has held since then
$\mathrm{P}\,\phi$	*previously*, ϕ held in some past event on the considered path
$\mathrm{H}\,\phi$	*historically*, ϕ held in every past event on the considered path

Operators $\mathrm{Y}, \mathrm{EY}, \mathrm{S}, \mathrm{AS}, \mathrm{ES}$ can be considered as primitive, while the other operators can be derived from them: $\mathrm{AY}\,\phi \triangleq \neg\,\mathrm{EY}\,\neg\phi$; $\mathrm{P}\,\phi \triangleq \top\,\mathrm{S}\,\phi$ (similarly for AP, EP with AS, ES); $\mathrm{H}\,\phi \triangleq \neg\,\mathrm{P}\,\neg\phi$ (similarly for AH, EH with EP, AP).

Some of the properties will hence be monotonic, in the sense that an intermediate verdict fixes all future verdicts, by quantification across paths (A/E) over the past (P/H).

From our work in [9], we present two example properties using the above past-CTL operators. Let us assume a property a_i indicates that a device (e.g. a drone) is in an area which we identify through its index i. Then the following formula will indicate success in the sense that all areas have been handled (assuming that drones come periodically into contact with each other so that knowledge propagates): $\bigwedge_{i\in N} \mathrm{EP}\,a_i$. A slightly more elaborate construction would be $\bigwedge_{i\in N} \mathrm{AH}\,\neg(a_i \wedge \mathrm{EY}(\mathrm{EP}\,a_i))$, which asserts that no device is servicing an area that has already been serviced by someone. In both cases we have the side-effect that every device updates and spreads the list of areas that have been handled with its neighbours in each round, propagating that knowledge through the network.

2.3 Spatial Logic: SLCS

The physical distribution of nodes in space over time (each node can be in proximity of different sets of nodes at any given time during its history) requires another dimension along which we would like to express interesting properties. Here, we have settled on Ciancia et al.'s Spatial Logic of Closure Space (SLCS) [20], which is based on two main modalities: $\Box\,\phi$ (holds in the interior of points where ϕ holds) and $\Diamond\,\phi$ (holds in the closure of points where ϕ holds). We have shown that this logic is able to express properties of discrete networks of devices [6]. A pictorial representation of this concept is given in Figure 2, where we have a set of nodes satisfying some property P, the *closure* $\Diamond P$ of all nodes adjacent to those satisfying P, and those where some P is reachable, $\mathcal{F}P$. Next, we present the local and global modalities of SLCS, which we will later map to equivalent constructs in AP (either as elementary functions or as macros on top of a functionally complete subset of operators).

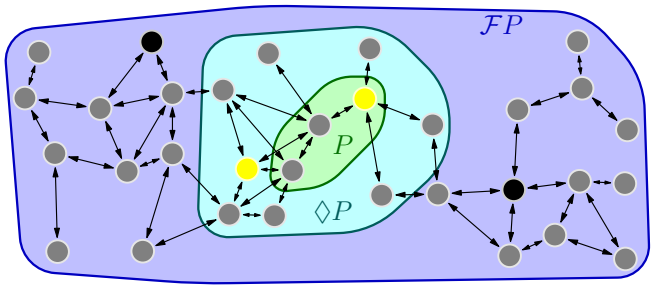

Fig. 2. Pictorial representation of a closure space induced by a graph [6]. Nodes are coloured black/grey/yellow according to which set of atomic propositions holds, edges are the proximity relation. Shaded areas satisfy (different closures of) some example property P.

Local modalities	
$\Diamond\,\phi$	*closure*, true at points with *some* neighbour satisfying ϕ
$\Box\,\phi$	*interior*, true at points with *all* neighbours satisfying ϕ
$\partial\,\phi$	*boundary*, true at points with some (but not all) neighbours satisfying ϕ
$\partial^-\,\phi$	*interior boundary*, true at points satisfying ϕ with some neighbour not satisfying it
$\partial^+\,\phi$	*closure boundary*, true at points not satisfying ϕ with some neighbour satisfying it
Global modalities	
$\phi\,\mathcal{R}\,\psi$	*reaches*, true at the ending points of paths whose starting point satisfies ψ and where ϕ holds
$\phi\,\mathcal{T}\,\psi$	*touches*, true at the ending points of paths whose starting point satisfies ψ and where ϕ holds in the rest of the path
$\phi\,\mathcal{U}\,\psi$	*surrounded by*, true at points in an area A satisfying ϕ, whose neighbours satisfy ψ
$\mathcal{G}\,\phi$	*everywhere*, true at points where ϕ holds in *every* incoming path
$\mathcal{F}\,\phi$	*somewhere*, true at points where ϕ holds in at least *one point* of an incoming path

Note that there is a functionally complete minimal set with $\Diamond$, $\mathcal{R}$ and Boolean operations which allows us to express all other operators as equivalences. Local modalities: $\Box\,\phi \equiv \neg\Diamond\neg\phi, \partial\,\phi \equiv (\Diamond\,\phi)\wedge\neg(\Box\,\phi), \partial^-\,\phi \equiv \phi\wedge\neg(\Box\,\phi), \partial^+\,\phi \equiv (\Diamond\,\phi)\wedge\neg\phi$. Global modalities: $\phi\,\mathcal{T}\,\psi \equiv \phi\,\mathcal{T}\,\Diamond\,\psi, \phi\,\mathcal{U}\,\psi \equiv \phi\wedge\Box\,\neg(\neg\psi\,\mathcal{R}\,\neg\phi), \mathcal{F}\,\phi \equiv \top\,\mathcal{R}\,\phi$.

As a sample application of SLCS in a smart home scenario, consider the following property to monitor: *air conditioning and lights are on whenever the room is not empty, off otherwise*. Consider the atomic propositions where *(i)* P is true on points that are sensing the presence of people; *(ii)* D is true on points that are the monitored electrical devices (air conditioning, lights); *(iii)* O is true on electrical devices that are on.

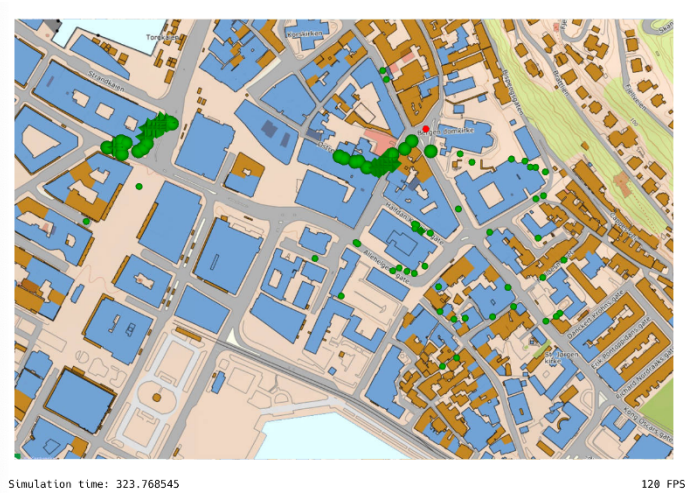

Fig. 3. Screenshot from an FCPP-simulation of the *cluster*-scenario, where node shape (circle/diamond), size (small/large) and colour (green/red) indicate a node's properties for *warning*, *cluster* and truth-value of the final property, respectively. The simulation is overlaid on top of a city-plan.

If we only want to consider the presence of people in the immediate vicinity, the considered property can be written as $\neg D \vee (O \Leftrightarrow \Diamond P)$. When also considering people farther away, the property can be written as $\neg D \vee (O \Leftrightarrow \mathcal{F} P)$. In the sample closure space of Fig. 2, a possible evaluation of these properties could be the following, where different colours are used for points where D is false (grey), D is true and O is false (black), D and O are true (yellow): The green area denotes the nodes for which P is true, i.e., the nodes that do perceive at least a person in the room. The cyan area ($\Diamond P$) is given by the nodes that have at least a neighbour within the green area. The blue area ($\mathcal{F} P$) is given by the nodes for which there exists a path to a node in P ("somewhere"); this includes all the nodes of the example.

In the practical session, we will use a more dynamic example where we combine AP properties with temporal and spatial properties: let us first assume the following two space-time variables: *warning* which is true if there are more than five neighbours in our direct vicinity, and another that indicates that a device is member of a *cluster* when at least three neighbours have the *warning*-flag set. We will later see that both flags can easily be derived through AP features that just rely on counting direct neighbours. Based on this, we can flag when we become part of a cluster by observing the transition of our corresponding variable through the (degenerate, as it is just per-device) past-CTL formula: $\mathit{cluster\_start} := (\mathrm{Y}\, \neg \mathit{cluster}) \wedge \mathit{cluster}$. Similarly, we define $\mathit{cluster\_end}$. Through an SLCS-formula a device can also observe if all its neighbours "agree" that they are part of a *cluster*: $\mathit{all\_cluster} := \mathcal{G}\, \mathit{cluster}$. And finally we can use this property in turn in a path-quantified past-CTL formula which states that

"no new neighbour joined since everyone observed formation of the cluster": $\neg cluster\_start$ AS $all\_cluster$. Fig. 3 graphically illustrates a given situation in the FCPP-simulator.

3 Aggregate Programming for Proximity-based Networks

Aggregate programming [15,38] is a computational paradigm for engineering distributed systems composed of large numbers of interacting devices, typically deployed in a physical or logical space and connected through neighbourhood relations. The paradigm has been developed to address the intrinsic complexity of systems such as the Internet of Things (IoT), pervasive computing infrastructures, and spatially situated cyber–physical systems, where devices operate asynchronously, may be mobile, and interact primarily through local communication. At its core, aggregate programming promotes a *global* view of computation: instead of describing individual device behaviours and their message exchanges explicitly, developers specify the intended collective behaviour of the system as a whole. Communication, synchronisation, and coordination among devices are abstracted away and handled implicitly by the computational model and its semantics. This approach enables programs to be expressed compositionally and declaratively, while remaining robust with respect to changes in network size, topology, and device density.

An aggregate program `P` is executed repeatedly and asynchronously by every device in the network. Each device δ follows its own local firing cycle, with no assumption of global synchronisation or shared clocks. A single execution of `P` on a device is referred to as a *computation round* (or *firing*), and can be informally described as consisting of the following phases:

1. **Context acquisition:** the device gathers contextual information, including sensor readings, locally stored state from the previous firing, and messages received from neighbouring devices since the last execution.
2. **Local evaluation:** the aggregate program `P` is evaluated using the acquired context as input, according to the operational semantics of the language.
3. **State update and communication:** the result of the evaluation is stored locally, possibly exposed to actuators, and may induce a broadcast to neighbouring devices for use in their subsequent firings.
4. **Sleep:** the device waits for its next round while listening for new messages.

Through the continuous repetition of these rounds across devices and over time, a coherent *global behaviour* emerges at the network level [37]. This can be formalised through the same notion of *Augmented Event Structure* introduced in Section 2.1, with one event for each round, connecting events $\epsilon_1 \rightsquigarrow \epsilon_2$ when the broadcast resulting from round ϵ_1 was considered in the context acquisition phase of round ϵ_2. This behaviour can be understood as the evolution of distributed data over space and time, rather than as the result of explicit message-passing protocols, and formalised accordingly as follows.

Definition 2 (Space–Time Value). *Given an augmented event structure* $\mathbb{E}$ *and a universe of values* $\mathbf{V}$*, a* space–time value *is a function* $f : E \to \mathbf{V}$*, associating a value with each event in* $\mathbb{E}$*. We denote as* $\mathbf{V}(\mathbb{E})$ *the set of such space–time values.*

From this perspective, an aggregate computation denotes a transformation of space–time values. This leads naturally to the notion of space–time functions.

Definition 3 (Space–Time Function). *An* n-ary space–time function *in an augmented event structure* $\mathbb{E}$ *is a partial function* $\mathtt{f} : \mathbf{V}(\mathbb{E})^n \twoheadrightarrow \mathbf{V}(\mathbb{E})$*, mapping tuples of input space–time values to an output space–time value defined over the same event structure.*

Not all space–time functions are implementable by aggregate programs: realizable functions must respect causality and be computable through local interactions [3]. Nevertheless, this abstraction provides a powerful semantic foundation for reasoning about distributed computations, linking the local execution of programs on individual devices with their global, collective meaning.

3.1 The eXchange Calculus

The *exchange calculus* (XC) [4,5] is a compact functional language that provides a foundational model for aggregate programming [15,38]. Its design goal is to express distributed collective computations at a high level, while abstracting away low-level concerns such as message passing, neighbourhood management, device identities, and network size. To this end, XC relies on a single communication primitive capable of expressing all forms of neighbourhood interaction, called *exchange*, and on the central data abstraction of *neighbouring values*.

A neighbouring value (or *nvalue* for short) is used to represent both information collected from nearby devices during a computation round, as well as information to send back to neighbour devices as result of an *exchange* routine. Formally, it is defined as a finite map from device identifiers to *local literals*, together with a distinguished default value, written as $\mathtt{w} = \ell[\delta_1 \mapsto \ell_1, \ldots, \delta_n \mapsto \ell_n]$, where δ_i, ℓ_i are neighbour devices with an explicitly associated value, and ℓ denotes the default local literal, used whenever no explicit value is associated to a neighbour device. Local values (such as simple numbers) are treated uniformly as nvalues with an empty mapping, written $\ell[]$. As a result, every value manipulated in XC can be regarded as a nvalue.

Neighbouring values are crucial in the exchange of information with neighbours. Given a nvalue, we can also retrieve the local value in it for the current device via built-in `self`(`w`), which when evaluated on device δ returns the local value $\ell = \mathtt{w}(\delta)$ stored in `w` for δ (which could be either the default or an explicit value). Besides these basic usages, the manipulation of nvalues comes in mostly two main forms. First, the point-wise lifting of local built-in operators to nvalues, applying the built-in independently for each device entry. For instance, $2[\delta_1 \mapsto 1, \delta_2 \mapsto -1] + 4[\delta_1 \mapsto 3, \delta_3 \mapsto 5]$ produces as

result $6[\delta_1 \mapsto 4, \delta_2 \mapsto 3, \delta_3 \mapsto 7]$ (the sums for δ_2 and δ_3 use default values 2 and 4, respectively). Second, nvalues can be *folded* into a single local value, via built-in `nfold(f,w)` which combines the values associated with the currently *aligned* neighbour devices using the binary function `f`. For instance, `nfold`$(+, 2[\delta_1 \mapsto 1, \delta_2 \mapsto -1]])$ in a context where the current device is δ_1 and the only aligned neighbour is δ_3 will produce 1 (value for the current device, always considered aligned) $+2$ (value for δ_3) $= 3[]$.

In XC, when evaluating any sub-expression, we consider as *aligned* only the neighbours that in their last round evaluated that *same sub-expression*, both as position in the AST (Abstract Syntax Tree of the overall program), and as stack frame of function calls. This *alignment* process is not only used in `nfold` operations, but it is also crucial to ensure that multiple *exchanges* of information can be compositionally combined, while automatically matching the information pertinent to each of them, without generating unpredictable interferences.

Such exchange of information is formalised through a single built-in function called *exchange*, with the following form:

`exchange(`$\mathtt{e}_i$`, (n) => return` $\mathtt{e}_r$ `send` $\mathtt{e}_s$`)`

where `return` $\mathtt{e}_r$ `send` $\mathtt{e}_s$ is syntactic sugar for pair construction `pair`$(\mathtt{e}_r, \mathtt{e}_s)$, to make explicit the role of the two expressions. Its evaluation proceeds as follows:

- First, expression $\mathtt{e}_i$ is evaluated into a local value $\ell[]$, to be used as a starting value for new interactions.
- Then, a nvalue `w` is built, mapping each aligned neighbour δ' to the last value shared by δ' for the current device δ in this exchange expression, using the ℓ just computed as default. Notice that if it is the first execution of `exchange` on the current device δ, the default ℓ also provides the value for the current device δ in `w` (otherwise, the value shared by δ for itself in its previous round is used).
- Expressions $\mathtt{e}_r$ and $\mathtt{e}_s$ then are evaluated by substituting `w` for `n` in their bodies, resulting in values $\mathtt{w}_r$ and $\mathtt{w}_s$ respectively.
- Finally, value $\mathtt{w}_r$ is returned as the value of the whole `exchange` expression in the program, and value $\mathtt{w}_s$ is sent to neighbours, who will use it in their subsequent rounds to construct their nvalue `w`.

This exchange construct is sufficiently general to model any pattern of communication and information sharing between neighbour devices as a defined function on top of it. In this tutorial, we will sometimes use one of these patterns called `nbr`, that can be defined as follows:

```
def nbr(init, value) {
  exchange(init, (n) => return n send value)
}
```

The `nbr` function sends to neighbours the provided argument `value` (which is a nvalue, so it may encode different values for different neighbours). Then, it collects from neighbours the values that they sent to the current device, building

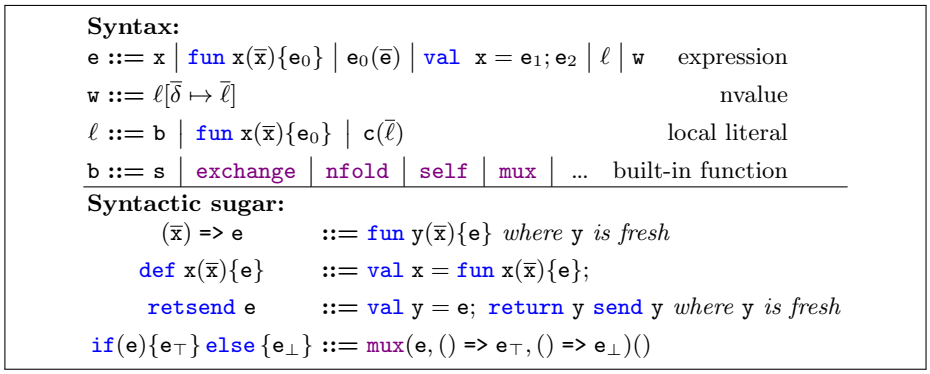

Fig. 4. Syntax of the eXchange Calculus (XC) language.

a neighbouring value `w` (using the value of `init` as default), which is returned by the whole function. Thus, the `nbr` function returns a view of the `value` argument on neighbours.

Figure 4 illustrates the syntax of expressions `e` in XC (we use $\overline{\texttt{x}}$ to denote a sequence of 0 or more elements), which can be either:

- a variable name `x`;
- a (possibly recursive) function definition, which may contain free variables;
- a function call, where the overline $\overline{\texttt{e}}$ is used to denote a sequence $\texttt{e}_1, \dots, \texttt{e}_n$;
- a let-like expression, where the value $\texttt{e}_1$ is substituted in $\texttt{e}_2$ for the free vriable `x`;
- a local literal ℓ, that is either the name of a built-in function `b`, a defined function *without* free variables, or a data constructor `c` applied to a number of local literals (possibly none);
- a *nvalue* `w`, as defined previously.

In order to ensure that XC programs are not dependent of a specific network configurations, nvalues are not allowed to directly appear in source programs, but only arise during expression evaluation. In addition to the basic syntax, we will also exploit some syntactic sugar (Figure 4, bottom) for expressing anonymous functions and defining named functions, for returning and sending a same expression in an `exchange`, as well as for `if`/`else` branching based on a condition. The latter exploits built-in $\texttt{mux}(\texttt{e}, \texttt{e}_\top, \texttt{e}_\bot)$, a multiplexer function returning either the value of $\texttt{e}_\top$ or $\texttt{e}_\bot$ depending on whether `e` is true or false. Note that expressions on different branches of an `if` *do not* align in the sense explained above (since they are in different parts of the AST).

Example 1 (Distance-To). Consider the following example of a function computing in each device its distance in hops from its closest *source*:

```
def dist(source) {
  exchange(∞, (d) =>
```

```
    retsend mux(source, 0, nfold(min, d, ∞)+1)
  )}
```

The `dist` function calculates hop-count distances from the nearest device where `source` is true using a single `exchange` construct. This construct collects distance estimates for neighbouring devices into a neighbouring value `w` (which maps neighbour devices to their distance estimates, defaulting to ∞ when no information is available), and assigns it to the variable `d`. The inner anonymous function body returns zero for source devices. For other devices, as in the traditional Bellman-Ford algorithm, it returns the minimum value from the neighbours' estimates `nfold(min, d, ∞)` increased by one.

This version of `nfold` with an additional argument works similarly as the `nfold` previously explained, except that it uses the value of its third argument (∞ in this case) as value for the current device, thus ignoring the value for the current device in the second argument (`d` in this case).

3.2 The FCPP Framework: a C++ DSL and Toolchain for AP

The XC programming language design has so far been implemented by several practical frameworks (see Section 1). In this tutorial, in order to experiment with space-time monitors we will use FCPP,[4] which is a C++-based domain-specific language and execution framework for aggregate programming. This framework provides a practical environment for implementing, simulating, and deploying aggregate programs on real distributed systems, ranging from simulations to networks of embedded devices. The framework can simulate asynchronous execution, message delays, communication failures, device mobility, and all kinds of dynamic changes in network topology and composition, making it suitable for modelling realistic distributed environments. FCPP is designed as a faithful implementation of the aggregate programming model and, in particular, of the semantics of the field calculus and the exchange calculus. Since it includes an implementation of both spatial and temporal logic operators within its standard library of aggregate algorithms, it is a natural choice for experimenting with the theory introduced in this tutorial.

A distinctive strength of FCPP lies in its versatility across different execution environments. The framework supports:

Interactive 3D graphical simulations, where large networks of devices are visualised in space and their aggregate state evolves in real time. This environment is particularly useful for debugging, teaching, and qualitative analysis of spatial behaviours.

Batch simulations with statistical analysis, enabling repeated runs under varying parameters and initial conditions. These simulations can automatically produce plots and aggregated metrics, supporting quantitative evaluation of algorithmic properties such as convergence time, resilience, and stability.

[4] Monitors used in this paper are available at: https://github.com/fcpp-experiments/past-ctl-monitoring

High-performance processing of large graph-based datasets, where aggregate programs are executed over massive static or dynamic network topologies in HPC environments. In this setting, FCPP can be used to analyse complex relational data through aggregate computations.

Deployment on real distributed systems, including networks of embedded devices, robots, or IoT nodes.

FCPP is designed so that the same aggregate programs can be reused on multiple execution environments, possibly with minimal adaptation, thanks to the separation between program logic and execution platform. These multiple usage modes make FCPP both a research tool for studying aggregate algorithms and a practical platform for engineering distributed collective systems. In this tutorial, we will exclusively employ the 3D graphical simulation environment. This setting provides immediate visual feedback on the evolution of space-time values and allows us to illustrate monitoring and runtime verification techniques in a controlled yet expressive scenario. The other execution modes remain fully compatible with the techniques discussed here, but fall outside the scope of the present exposition.

At the language level, FCPP realises aggregate constructs through templated C++ functions and classes. The library provides macros and constructs to enable expressing programs with a syntax closely resembling that of XC, while allowing the underlying runtime system to manage communication, state persistence, and alignment of sub-computations, as shown in this code snippet.

```
FUN bool monitor(ARGS, int source_id) { CODE
    bool close = dist(CALL, node.uid == source_id) < 10;
    field<bool> neigh_close = nbr(CALL, close);
    return all_hood(CALL, neigh_close);
}
```

Several macros are used in this snippet. `FUN` hides a template argument for the type of the node. `ARGS` hides arguments for the necessary execution context (the `node` object, and a representation of the stack trace). `CODE` hides updating of the stack trace with the current function call. `CALL` passes the execution context along in function calls. The `node` object is not only implicitly used within constructs such as `nbr` or `all_hood` (which in XC would be `nfold` with logical and), but also explicitly in order to directly access to sensors and actuators available in the node (such as `node.uid` in the example above, to access a unique identifier for the node). This code snippet also shows the usage of neighbouring values, which are represented through a dedicated `field<T>` class, with built-in operators providing support for both local value extraction, as well as point-wise and folding operations.

$\top$	`true`	q	`q()`	$\neg\phi$	`!`ϕ	$\phi_1 \vee \phi_2$	ϕ_1 `|` ϕ_2
$\Diamond\,\phi$	`nfold(|, nbr(false,` ϕ`))`			$\phi_1\,\mathcal{R}\,\phi_2$	`if (`ϕ_1`) {dist(`ϕ_2`)<D} else {false}`		
$\mathrm{Y}\,\phi$	`self(nbr(false,` ϕ`))`			$\mathrm{EY}\,\phi$	`nfold(|, nbr(false,` ϕ`))`		
$\phi_1 \mathrm{S}\, \phi_2$	`exchange( false, (old) => retsend` ϕ_2 `| (`ϕ_1 `& self(old)))`						
$\phi_1 \mathrm{AS}\, \phi_2$	`exchange( false, (old) => retsend` ϕ_2 `| (`ϕ_1 `& nfold(&, old)))`						
$\phi_1 \mathrm{ES}\, \phi_2$	`exchange( false, (old) => retsend` ϕ_2 `| (`ϕ_1 `& nfold(|, old)))`						

Fig. 5. Translation of a primitive set of SLCS and past-CTL operators into XC.

4 Monitors for past-CTL and SLCS Formulas

4.1 Generating XC Monitors

Formulas of SLCS and past-CTL logics can be translated into XC Boolean expressions, and then evaluated during the execution of a program. Figure 5 shows a possible translation, by recursion on sub-formulas. We translate atomic propositions q into function calls `q()`, which produce a result by accessing the readings of sensors and/or performing simple local computations. Logical formulas without temporal/spatial modalities are easily translated by mapping the logical operators to their XC representations. In Figure 5 (first line), we show the translations of the operators $\neg$ and $\vee$.

For SLCS we consider the translation of primitive modalities $\Diamond$ and $\mathcal{R}$ (second line in Figure 5). The representation of formula $\Diamond\,\phi$ should be understood as follows:

- the most recent Boolean values of ϕ received by the neighbours (including the current device) are collected into an nvalue with `nbr`
- the folding operator `nfold` applies logical "or" to the nvalue, returning true iff at least one of the elements in the nvalue is true

The representation of formula $\phi_1\,\mathcal{R}\,\phi_2$ requires a function `dist`, and a value `D`, where:

- `dist(`ϕ_2`)` performs an aggregate computation of the distance between the current device and the closest device where ϕ_2 holds (see Section 3.1)
- `D` is an upper bound to the network diameter (either fixed at design time or estimated with an aggregate computation)

Notice that, as function `dist` is executed within the `if` branch where ϕ_1 is true, it will only receive messages from neighbours for which ϕ_1 is also true. Thus, that function call computes the shortest distance from a point satisfying ϕ_2, restricted within the region where ϕ_1 is true. The XC expression for formula $\phi_1\,\mathcal{R}\,\phi_2$ evaluates to true if such a distance is within the network (i.e., `< D`), to false otherwise.

Regarding formal guarantees, [37] shows that each monitor self-stabilises to the value of the abstract formula over the limit topology of the device network, and specific bounds on convergence time have also been obtained in [10].

For Past-CTL, we consider the translation of primitive modalities Y, EY, S, ES, and AS. The representation of $\mathrm{Y}\,\phi$ extracts with `self` the value for the current device from the nvalue obtained by exchanging the Boolean value of ϕ with neighbours. In other words, we are taking the value of ϕ at the previous round on the current device. Interestingly, the translation of $\mathrm{EY}\,\phi$ is the same as that of $\Diamond\,\phi$ discussed above, since *neighbour devices* can only be accessed through *previous events*. The representation of formula $\phi_1\,\mathrm{S}\,\phi_2$ is a bit longer:

- the `old` argument of the lambda function passed to `exchange`, receives the most recent Boolean values of the formula computed by the neighbours (including the current device)
- the value of the formula in the current device is true, either if: ϕ_2 is currently true; or, the S formula was true on the device in the previous round (the value is obtained with `self` from `old`), and ϕ_1 is currently true

The translations of $\phi_1\,\mathrm{ES}\,\phi_2$ and $\phi_1\,\mathrm{AS}\,\phi_2$ are very similar to the one we just describe, except that they require that the formula was true in at least one, (resp. all) neighbours in the previous round.

Interestingly, for past-CTL the monitors correspond exactly to the associated formulas, i.e., they *always* return the correct values.

4.2 FCPP Implementation

Each past-CTL and SLCS operator has been implemented in FCPP as an aggregate function, in the namespace `fcpp::coordination::logic`. For example, the Y past-CTL operator is implemented as `FUN bool Y (ARGS, bool f)` , where `FUN` and `ARGS` are C++ macros explained in Section 3.2, and the argument `f` takes the Boolean value of the formula ϕ to which Y is applied, as computed in the current round and device. Clearly, since XC and its FCPP translation are in *functional* style, logic operators with two or more arguments are applied using *prefix* instead of *infix* notation. For example, to evaluate formula $\phi_1\,\mathcal{R}\,\phi 2$, the following calls are issued:

```
f1 = ...; f2 = ...;
f = R(CALL, f1, f2);
```

We point out that, although we have shown how all past-CTL and SLCS operators can be realized in terms of a few operators chosen as *primitive*, FCPP implements each operator from scratch, to improve efficiency.

Among the operators of Past-CTL and SLCS, the ones that offer a wider range of possible implementations are the global SLCS operators $\mathcal{R}$, $\mathcal{T}$, $\mathcal{U}$, $\mathcal{G}$, and $\mathcal{F}$. The reason is that, as seen for the $\mathcal{R}$ primitive operator above, they all depend on a `dist()` function which can be implemented in several ways, trading-off the network convergence time, the resources overhead, and other factors (see [1] for details).

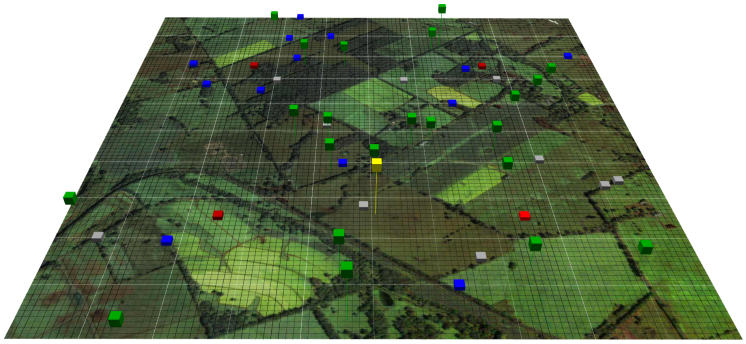

Fig. 6. Screenshot of the FCPP simulation of the drones recognition example.

4.3 Experimenting with the FCPP Framework

In an example scenario implemented in the FCPP simulator[5], we defined an outdoor area subdivided into $N = 4$ square areas, which had to be handled (e.g., observed with a camera) by a given set of 50 drones (Figure 6). Each subarea had a communication tower at its middle (brown/red cubes), and drones swept the environment performing random tasks (green cubes are drones heading to a destination, yellow ones are handling their goal), interleaved by recharging (blue) and waiting for tasks (grey) periods. The four towers issued "handling"-requests at random times, instructing the closest waiting drones to reach their area.

Proposition a_i for $i = 1 \dots N$ is true iff the current drone is now handling area i. The properties below are formulated in terms of those observables, and how drones actually move or communicate are external factors for the purpose of our example. The existential quantifiers also encode that at the same point in time, different drones may have different views of the global system. Consider the properties that have already been discussed in Section 4: (i) every area gets eventually handled by a drone – $\bigwedge_{i \in N} \mathrm{EP}\, a_i$; (ii) no drone is handling an area that it knows to be already handled – $\bigwedge_{i \in N} \mathrm{AH}\, \neg(a_i \wedge \mathrm{EY}(\mathrm{EP}\, a_i))$.

The monitoring of the first property in FCPP is based on function:

```
FUN bool logic::area_handled(ARGS, bool handling) { CODE
    return EP(CALL, handling);
}
```

That is, we have an aggregate function `logic::area_handled` which takes as input a Boolean value telling whether *an* area is being *handled now* by the current drone. And returns true iff this drone, or any other drone which has already been able to propagate the information, has handled the area up to now. Interestingly, the calls to `logic::area_handled` are issued within a `LOOP` macro, which

[5] Introduced in paper [9], and available with other examples on GitHub, at https://github.com/fcpp-experiments/past-ctl-monitoring

guarantees through alignment (Section 3.1) that the information for each area is kept separate (i.e., if a drone handles area i this does not affect the value of `logic::area_handled` for area $j \neq i$). Finally, the values of the function call for each area are simply $\wedge$-ed to get the value of the complete property.

The monitoring of the second property in FCPP is based on function:

```
FUN bool no_redundancy(ARGS, bool handling) { CODE
    return AH(CALL, !(handling & EY(CALL, EP(CALL, handling))));}
```

While the property is more complex, the implementation follows the same lines as that of the first property.

Simulations. The concern of (visually appealing) simulation and gathering statistics from them is orthogonal to the challenges of programming with FCPP, with or without monitors. While visualization will always have to be customized for the scenario at hand, the FCPP library offers handy operations to specify scenarios, such as deploying a group of agents of a particular type with a certain distribution in a starting area for that group, and FCPP fragments for group members following a leader on a programmed or random walk.

5 Case Studies in AP and Challenges for AP-based DRV

In this section, we briefly examine some industrial case studies that we have addressed with the FCPP framework, and use them to identify interesting challenges for AP-based DRV, focusing in particular on the need for more expressive monitor specification logic. Note that we have *not yet* implemented any DRV functionality in the use cases. However, incorporating it is expected to require negligible effort compared to the development of the AP algorithms, and even more so relative to implementing simulation scenarios or the software modules for physical devices. Deployments on physical devices have been limited in scale due to costs and other practical constraints. Simulations using the Gazebo simulator allowed us to increase the scale (e.g., up to eight Jackal robots in the Torino Airport outdoor use case), but they were still constrained by Gazebo's high computational requirements. In contrast, lightweight simulations within FCPP enable scaling to hundreds of thousands of devices on a high-end laptop.

5.1 Some Industrial Case Studies in FCPP

Torino Airport Indoor. In the RoboApp project [35], SAGAT (the company that manages Torino Airport, https://www.aeroportoditorino.it/en) proposed a compelling use case, involving the boarding gates instead of outdoor areas. In particular, it can happen that, due to several reasons, long queues of people form at the gates, overflowing out of the serpentine barriers into the transit areas. Such queues should be promptly managed by human operators, since they get in the way of people passing through, but it is often the case that queues form while no

Fig. 7. Queue detection with the RoboAPP CV pipeline.

one is (yet) attending the gate. We proposed to exploit a team of Autonomous Mobile Robots (AMRs) in charge of monitoring the gates, equipped with sensors and software for navigating and observing the environment with a camera, as well as with an FCPP application for distributed control and coordination. When the schedule (which is currently fixed) requires that a gate is monitored, the team executes a distributed XC algorithm to choose two (or more) robots that will jointly execute the task. The chosen robots navigate from the parking areas (where they gather when idle) to predefined locations close to the gate, where they can start their observations.

Detecting a queue near a gate (see Figure 7) is a challenging computer-vision task, requiring a non-trivial (Computer Vision) CV pipeline executed on-board Jetson Nano computers. Each robot perceives the scene from its own point of view, and may draw different conclusions about the presence of an overflowing queue. An FCPP application running onboard, implements a distributed XC algorithm to reach a consensus among the robots (e.g., a distributed majority voting). However, since even the consensus among multiple points of view is sometimes unreliable, we have added an additional component to the system, namely an AI Agents [39] application running on a PC on the local network, based on a multi-modal Small Language Model (SLM). The AI application is invoked to make a final decision when the team of robots agrees that there is *likely* a critical situation to notify to the human operators.

Torino Airport Outdoor. The RoboNG project [31] addressed another use case supplied by SAGAT, which proposed a relevant problem faced by airports, namely the timely detection and removal of FODs (Foreign Object Debris). FODs are small objects that can clutter the parking and taxiing areas used by airplanes, with potentially disastrous effects. Continuously inspecting the relevant areas for FOD detection is a repetitive, costly, and error-prone task when

Fig. 8. The Jackal by Clearpath (left), the Create3 by iRobot (center) equipped for RoboNG; a Sewio UWB anchor (right).

performed by humans. The RoboNG project proposed the prototype implementation of a system able to perform FOD detection in a (semi) autonomous way.

In particular, a team of Autonomous Mobile Robots (AMRs) is charged with covering the relevant areas while exploiting on-board sensors to detect potential FODs, and send alerts to human operators. In order to be able to navigate the environment, perceive it, and communicate with the operator each robot is equipped with a high-resolution camera, a CV pipeline for the detection of FODs, and autonomous navigation software (supporting obstacle and collision avoidance), powered by a Lidar and/or an additional camera. The AMRs communicate with each other via a WiFi connection to a local wireless network supplied by a router, and host on-board a FCPP application that enables communication and coordination among robots. As the main *robot brain*, the FCPP application gets information from the CV pipeline, and accesses the local robot to get telemetry and send control commands. Figure 8 shows the two robots employed for the prototype system implementation. The Jackal, produced by Clearpath Robotics, is a powerful small-size rover for outdoor exploration. The Create3, produced by iRobot, is a basic platform for research, usable mainly indoor. The distributed covering algorithm was implemented as an XC program running as an on-board FCPP-based application.

Smart Agriculture. Among the many partners of the Agritech National Research Center for Technology in Agriculture, Spoke 6 (https://agritechcenter.it/), we have cooperated in particular with the Department of Agricultural, Forest and Food Sciences (DISAFA) of the University of Turin. Most relevant here, is the study of the employment of swarms of drones in agriculture. Guided by the domain experts at DISAFA, we have implemented several simulations of swarms of autonomous drones, controlled and coordinated with AP algorithms implemented in XC. We have focused in particular on the following problem: given an autonomous vehicle (either aerial or ground), named *worker*, in charge of following a pre-determined trajectory across a crops area to perform its task

(e.g., weeding, or trimming), devise algorithms to coordinate a swarm of drones that must accompany the worker along its trajectory providing surveillance of the surrounding areas, for safety purposes. An important variant that we have explored, consists in introducing multiple workers: the problem becomes much more interesting (from the AP point of view) when each worker of a given set has its associated swarm, and individual drones can move from one swarm to another one to optimize monitoring. In [29], we have reported the results of a cooperation with DISAFA, where we have experimentally studied the effectiveness of the implementation based on a single worker and lightweight simulation in providing surveillance for a UGV in a vineyard, taking into account the detailed 3D map of the terrain and the field-of-view of the cameras on the drones.

The MAcA Living Lab. The MAcA Living Lab was established at the MAcA Environmental Museum (https://www.acomeambiente.org/en/) in Turin, Italy, operating within the framework of the European NODES Project, Spoke 4 "Digital Innovation Toward Sustainable Mountain". The museum encompasses approximately 3,500 square meters of indoor and outdoor space, and is an ideal setting for exploring BIM-centric Digital Twins for sustainable cultural and educational facilities. The museum has already been equipped with a heterogeneous IoT sensor network, including sensors for: weather, humidity, air quality, indoor parameters (e.g., temperature and CO_2), and presence/activity detection. More recently, we have started to deploy an RTLS (Real-Time Localization System) based on Ultra-wideband (UWB) anchors – Figure 8 (right) – and tags, to be complemented also by UWB-enabled smartphones. We will exploit XP distributed algorithms both to optimize real-time localization through cooperation, and to perform on-device aggregate computation of spatial-dependent measures (e.g., crowd density) and collective behavior analysis.

5.2 Challenges for AP-based DRV

Past-CTL and SLCS provide a solid logical foundation for aggregate-based DRV, applied to complex collective systems such as those outlined in Section 5.1. Crucially, runtime verification becomes increasingly important with the introduction of AI modules in those systems, as it can make them less deterministic and less reliable. From our case studies, several expressive and methodological limitations or the presented logics emerge, that highlight important research challenges and suggest directions for future investigation.

A first limitation concerns the expressive power of *past-CTL*. By construction, past-CTL only allows reasoning over the past evolution of space-time values. This restriction prevents the specification of inherently forward-looking properties such as eventual convergence, stabilization guarantees, or liveness conditions. On the other hand, it is a natural limitation for runtime monitoring, where future behaviour is unknown and only the past is monitorable. Future work may devise a predictive semantics and possibly automatic translations of (certain classes of) forward-looking properties (in logics such as CTL) to past-CTL, closing this gap.

In the Torino Airport Outdoor case study, we could e.g., require that "every cell will eventually be visited", which is the main goal of the swarm in that scenario.

A second issue lies in the limited interaction between temporal and spatial dimensions. In past-CTL, spatial reasoning is mediated through aggregate constructs, but the logic does not offer first-class operators for structured spatial quantification (e.g., reasoning over dynamically defined regions, boundaries, or topological features). Conversely, *SLCS* provides a robust spatial logic based on closure and neighbourhood relations, yet it lacks intrinsic temporal expressiveness. When used in combination, the two logics still exhibit a conceptual separation between time and space, by also relying on two incompatible semantic spaces, making it difficult to express genuinely spatio-temporal patterns such as moving fronts, persistent corridors, or region-wide synchronization phenomena. In the Torino Airport Indoor case study, we could e.g., require that when a gate opens, a robot should be available and able to reach it. A more integrated spatio-temporal logic for aggregate systems remains an open research direction.

Another challenge concerns quantitative reasoning. Both past-CTL and SLCS are primarily qualitative: they express whether a property holds, but not to what degree, with what robustness margin, or under which quantitative constraints. However, many aggregate applications require reasoning about numerical thresholds, convergence rates, probabilistic guarantees, or robustness against perturbations. Quantitative semantics could be incorporated in many different ways, including: spatio-temporal operators with metric bounds, counting constraints, or parametric first-order predicates. Each of these extensions would significantly enhance the applicability of AP-based DRV. For example, in the Smart Agriculture case study, we could require that two workers (e.g., UGVs) never stay closer than R meters for more than a period of time of T seconds.

An additional and largely unexplored direction concerns extending spatio-temporal logics to explicitly reason about *graph edges*, rather than only node-based properties. Both past-CTL and SLCS interpret formulas over devices, with spatial operators implicitly relying on adjacency, yet many aggregate behaviours are inherently edge-centric: links may carry weights or reliability annotations, may fail dynamically, and often determine properties such as flow consistency, cut detection, or path optimality. Supporting edge predicates and mixed node–edge formulas would therefore broaden the range of specifiable behaviours. Such an extension would also promote higher-level graph abstractions to first-class logical entities: for instance, *regions* could be characterised through connectivity constraints, and *shortest paths* could be captured directly through edge-aware predicates. In the MAcA Living Lab case study, we could require that each person can follow a path of wireless point-2-point links to an emergency exit, without relying on the wired network being operational. Enriching the logic with such custom graph predicates could be done by leveraging the graph-theoretic structure of aggregate computations, but defining a coherent semantics for this while preserving locality and monitorability remains an open challenge.

Acknowledgments. This publication is part of the project NODES, which has received funding from the MUR – M4C2 1.5 of PNRR funded by the European Union-

NextGenerationEU (Grant agreement no. ECS00000036). This study has been supported by the Italian PRIN project "CommonWears" (2020HCWWLP) and was carried out within the Agritech National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1032 17/06/2022, CN00000022). Volker Stolz is supported by Norwegian Research Council project CroFlow: Enabling Highly Automated Cross-Organisational Workflow Planning (grant no. 326249) and EU HORIZON MSCA Doctoral Network "Towards an Understanding of Artificial Intelligence (TUAI)" (grant no. 101168344). This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Aguzzi, G., Audrito, G., Viroli, M.: Optimising aggregate monitors for spatial logic of closure spaces properties. In: Proc. of the 7th ACM Int. Workshop on Verification and Monitoring at Runtime Execution. p. 25–31. VORTEX 2024, ACM (2024). https://doi.org/10.1145/3679008.3685544
2. Audrito, G.: FCPP: an efficient and extensible field calculus framework. In: Proceedings of the 1st International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS. pp. 153–159. IEEE Computer Society (2020). https://doi.org/10.1109/ACSOS49614.2020.00037
3. Audrito, G., Beal, J., Damiani, F., Viroli, M.: Space-time universality of field calculus. In: Coord. Models and Languages. Lecture Notes in Comp. Science, vol. 10852, pp. 1–20. Springer (2018). https://doi.org/10.1007/978-3-319-92408-3_1
4. Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M.: Functional programming for distributed systems with XC. In: 36th European Conference on Object-Oriented Programming, ECOOP 2022. LIPIcs, vol. 222, pp. 20:1–20:28. Schloss Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.20
5. Audrito, G., Casadei, R., Damiani, F., Salvaneschi, G., Viroli, M.: The exchange calculus (XC): A functional programming language design for distributed collective systems. J. Syst. Softw. **210** (2024). https://doi.org/10.1016/J.JSS.2024.111976
6. Audrito, G., Casadei, R., Damiani, F., Stolz, V., Viroli, M.: Adaptive distributed monitors of spatial properties for cyber-physical systems. J. Syst. Softw. **175**, 110908 (2021). https://doi.org/10.1016/j.jss.2021.110908
7. Audrito, G., Casadei, R., Damiani, F., Viroli, M.: Computation against a neighbour: Addressing large-scale distribution and adaptivity with functional programming and Scala. Log. Methods Comput. Sci. **19**(1) (2023). https://doi.org/10.46298/lmcs-19(1:6)2023
8. Audrito, G., Damiani, F., Rinaldi, S., Tagliabue, L.C., Testa, L., Torta, G.: Aggregate Programming for Customized Building Management and Users Preference Implementation, pp. 147–172. Springer International Publishing, Cham (2023). https://doi.org/10.1007/978-3-031-15160-6_7
9. Audrito, G., Damiani, F., Stolz, V., Torta, G., Viroli, M.: Distributed runtime verification by past-CTL and the field calculus. J. Syst. Softw. **187**, 111251 (2022). https://doi.org/10.1016/j.jss.2022.111251

10. Audrito, G., Damiani, F., Torta, G.: Real-time guarantees for SLCS monitors in XC. In: VORTEX Proceedings of VORTEX 2024. pp. 32–37. ACM (2024). https://doi.org/10.1145/3679008.3685545
11. Audrito, G., Rapetta, L., Torta, G.: Extensible 3d simulation of aggregated systems with FCPP. In: Coordination Models and Languages - 24th International Conference, COORDINATION 2022 Proceedings. LNCS, vol. 13271, pp. 55–71. Springer (2022). https://doi.org/10.1007/978-3-031-08143-9_4
12. Audrito, G., Terraneo, F., Fornaciari, W.: FCPP+Miosix: Scaling aggregate programming to embedded systems. IEEE Trans. Parallel Distributed Syst. **34**(3), 869–880 (2023). https://doi.org/10.1109/TPDS.2022.3232633
13. Audrito, G., Torta, G.: FCPP to aggregate them all. Sci. Comput. Program. **231**, 103026 (2024). https://doi.org/10.1016/J.SCICO.2023.103026
14. Audrito, G., Viroli, M., Damiani, F., Pianini, D., Beal, J.: A higher-order calculus of computational fields. ACM Trans. Comput. Logic **20**(1), 5:1–5:55 (2019). https://doi.org/10.1145/3285956
15. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the Internet of Things. IEEE Computer **48**(9) (2015). https://doi.org/10.1109/MC.2015.261
16. Casadei, R., Aguzzi, G., Pianini, D., Viroli, M.: Programming (and learning) self-adaptive & self-organising behaviour with ScaFi: for swarms, edge-cloud ecosystems, and more. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2023, Toronto, Canada, September 25-29, 2023. pp. 33–34. IEEE (2023). https://doi.org/10.1109/ACSOS-C58168.2023.00032
17. Casadei, R., Fortino, G., Pianini, D., Placuzzi, A., Savaglio, C., Viroli, M.: 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 (2022). https://doi.org/10.1109/JIOT.2022.3172470
18. Casadei, R., Viroli, M.: Towards aggregate programming in Scala. In: 1st PMLDC Workshop. pp. 5:1–5:7. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2957319.2957372
19. Casadei, R., Viroli, M., Aguzzi, G., Pianini, D.: Scafi: A Scala DSL and toolkit for aggregate programming. SoftwareX **20**, 101248 (2022). https://doi.org/10.1016/j.softx.2022.101248
20. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Specifying and verifying properties of space. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) Theoretical Computer Science. pp. 222–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44602-7_18
21. Cortecchia, A.: Multiplatform self-organizing systems through a kotlin-mp implementation of aggregate computing. In: 2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). pp. 155–157 (2024). https://doi.org/10.1109/ACSOS-C63493.2024.00048
22. Dustdar, S., Pujol, V.C., Donta, P.K.: On distributed computing continuum systems. IEEE Transactions on Knowledge and Data Engineering **35**(4), 4092–4105 (2023). https://doi.org/10.1109/TKDE.2022.3142856
23. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime Verification for Decentralised and Distributed Systems, pp. 176–210. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
24. Gigante, N., Montanari, A., Reynolds, M.: A one-pass tree-shaped tableau for ltl+past. In: LPAR-21. EPiC Series in Computing, vol. 46, pp. 456–473. EasyChair (2017). https://doi.org/10.29007/3HB9
25. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563

26. Leucker, M., Schallhart, C.: A brief account of runtime verification. The Journal of Logic and Algebraic Programming **78**(5), 293–303 (2009). https://doi.org/10.1016/j.jlap.2008.08.004
27. Mamei, M., Zambonelli, F., Leonardi, L.: Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. In: Petta, P., Tolksdorf, R., Zambonelli, F. (eds.) Engineering Societies in the Agents World III (ESAW 2002). LNCS, vol. 2577, pp. 68–81. Springer (2002). https://doi.org/10.1007/3-540-39173-8_6
28. Nitti, L., Bortoluzzi, D., Biglia, A., Aimonino, D.R., Torta, G., Audrito, G., Damiani, F., Gay, P., Rapp, A., Comba, L.: Drone-swarm based surveillance system for autonomous machine safety functionality. In: Precision agriculture '25. LNCS, vol. 2577, pp. 492–598. Brill (2002). https://doi.org/10.1163/9789004725232_077
29. Nitti, L., Bortoluzzi, D., Biglia, A., Aimonino, D.R., Torta, G., Audrito, G., Damiani, F., Gay, P., Rapp, A., Comba, L.: Drone-swarm based surveillance system for autonomous machine safety functionality. In: Precision agriculture'25, pp. 592–598. Wageningen Academic (2025). https://doi.org/10.1163/9789004725232_077
30. Saint-Martin, L., Delesse, J.P., Tual, J.P., Coulon, O., Roncière, J.C.d.l., Nana, L., Lebon, C.: Study on the economic potential of far edge computing in the future smart Internet of Things. Final study report, European Commission: Decision Etudes & Conseil, Directorate-General for Communications Networks, Content and Technology (November 2023). https://doi.org/10.2759/05608
31. Santer REPLY SpA: The RoboNG project. https://ecs-nodes.eu/en/1-aerospace-and-sustainable-mobility/progetti-imprese/robong (2023), last accessed 9 February 2026
32. Testa, L., Audrito, G., Damiani, F., Torta, G.: Aggregate processes as distributed adaptive services for the industrial internet of things. Pervasive Mob. Comput. **85**, 101658 (2022). https://doi.org/10.1016/j.pmcj.2022.101658
33. University of Bologna: ScaFi Aggregate Programming Toolkit. https://scafi.github.io/ (since 2018), last accessed 9 February 2026
34. University of Bologna: Collektive. https://collektive.github.io (since 2024), last accessed 9 February 2026
35. University of Turin: The RoboAPP project. https://ai4future.unito.it/roboapp/ and https://ecs-nodes.eu/en/1-aerospace-and-sustainable-mobility/progetti-accademici/roboapp (2024), last accessed 9 February 2026
36. University of Turin: FCPP. https://fcpp.github.io/ (since 2021), last accessed 9 February 2026
37. Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D.: Engineering resilient collective adaptive systems by self-stabilisation. ACM Trans. Model. Comput. Simul. **28**(2), 16:1–16:28 (2018). https://doi.org/10.1145/3177774
38. Viroli, M., Beal, J., Damiani, F., Audrito, G., Casadei, R., Pianini, D.: From distributed coordination to field calculus and aggregate computing. J. Log. Algebraic Methods Program. **109** (2019). https://doi.org/10.1016/j.jlamp.2019.100486
39. Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., et al.: A survey on large language model based autonomous agents. Frontiers of Computer Science **18**(6), 186345 (2024). https://doi.org/10.1007/s11704-024-40231-1
40. Winskel, G.: Event structure semantics for CCS and related languages. In: Nielsen, M., Schmidt, E.M. (eds.) Automata, Languages and Programming, 9th Colloquium, Aarhus, Denmark, July 12-16, 1982, Proceedings. LNCS, vol. 140, pp. 561–576. Springer (1982). https://doi.org/10.1007/BFB0012800