The Journal of Systems & Software 187 (2022) 111251

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Distributed runtime verification by past-CTL and the field calculus ☆

Giorgio Audrito [a,*], Ferruccio Damiani [a], Volker Stolz [b], Gianluca Torta [a], Mirko Viroli [c]

[a] Università di Torino, Turin, Italy
[b] Western Norway University of Applied Sciences, Bergen, Norway
[c] Alma Mater Studiorum—Università di Bologna, Cesena, Italy

ARTICLE INFO

Article history:
Received 17 January 2022
Accepted 27 January 2022
Available online 4 February 2022

MSC:
00-01
99-00

Keywords:
Distributed systems
Runtime verification
Field calculus
Temporal logic

ABSTRACT

Recent trends in the engineering of software-intensive systems increasingly promote the adoption of computation at the edge of the network, in the proximity of where sensing and actuation are performed. Applications are executed directly in IoT devices deployed in the physical environment, possibly with the aid of edge servers: there, interactions are essentially based on physical proximity, and communication with the cloud is sporadic if not absent.

The challenge of monitoring the execution of such system, by relying on local interactions only, naturally arises. We address this challenge by proposing a rigorous approach to distributed runtime monitoring for space-based networks of devices. We introduce the past-CTL logic, an extension of past-LTL able to express a variety of properties concerning the knowable past of an event. We formally define a procedure to derive, from a past-CTL formula, monitors that can be distributed on each device and whose collective behaviour verifies the validity of the formula at runtime across space and time. This is achieved by relying on the field calculus, a core programming language used to specify the behaviour of a collection of devices by viewing them as an aggregate computing machine, carrying out altogether a distributed computational process. The field calculus is shown to be a convenient language for our goals, since its functional composition approach provides a natural way of translating in a syntax-directed way properties expressed in a given logic into monitors for such properties. We show that the monitor process executing in each single device runs using local memory, message size, and computation time that are all linear in the size of the formula (1 bit per temporal connective). This matches the efficiency of the best available previous results for (non-distributed) monitors derived from past-LTL formulas. Finally, we empirically evaluate the applicability of the approach to sample problems in distributed computing, through simulated experiments with monitors written through a C++ library implementing the field calculus programming constructs.

1. Introduction

Runtime verification is a computing analysis paradigm based on observing a system at runtime (to check its expected behaviour) by means of monitors generated from formal specifications (Leucker and Schallhart, 2009). Distributed runtime verification is runtime verification in connection with distributed systems: it comprises both monitoring of distributed systems and usage of distributed systems for monitoring. Being a verification technique, additionally, runtime verification promotes the generation of monitors from formal specifications, so as to precisely state the properties to check as well as providing formal guarantees about the results of monitoring. Distribution is hence a particularly challenging context in verification, for it requires to correctly deal with aspects such as synchronisation, faults in communications, possible lack of unique global time, and so on Francalanza et al. (2018). Additionally, the distributed system whose behaviour is to be verified at runtime could emerge from modern application scenarios like the Internet-of-Things (IoT), Cyber–Physical Systems (CPS), Edge Computing (EC) or large-scale Wireless Sensor Networks (WSN). In this case additional features are to be considered, like openness (the set of nodes is dynamic), large-scale (a monitoring strategy may need to scale from few units up to thousands of devices), and interaction locality (nodes may be able to communicate only with a small neighbourhood). So, in the most general case, distributed runtime verification challenges the way in which one can express properties on such dynamic distributed systems, can express flexible computational tasks, and can reason about compliance of properties and corresponding monitoring behaviour.

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: giorgio.audrito@unito.it (G. Audrito), ferruccio.damiani@unito.it (F. Damiani), volker.stolz@hvl.no (V. Stolz), gianluca.torta@unito.it (G. Torta), mirko.viroli@unibo.it (M. Viroli).

https://doi.org/10.1016/j.jss.2022.111251
0164-1212/

In this paper we considered distributed runtime verification for systems where a significant portion of computations is performed at the edge of the network, namely, where computational devices have a (possibly changing) physical position and space, and interact locally, with devices in the proximity. As a notable example, used throughout the paper (among others thoroughly discussed in Section 4), we will refer to a synthetic logical network comprising nodes of different nature (cloud, fog, and edge), sporadically issuing requests (of a few types) and then waiting for a corresponding response. In this context, we will show how significant properties of the system can be expressed and monitored.

Developing on our preliminary findings reported in Audrito et al. (2018b), in this paper we argue that a promising approach to address the challenges of distributed runtime verification for these kinds of system (Shi et al., 2016) can be rooted on the computational paradigm of aggregate computing (Beal et al., 2015), along with the field calculus language (Audrito et al., 2019). Aggregate computing promotes a view of a distributed system as a conceptually single computing device, spread throughout the (physical or virtual) space in which nodes are deployed, that has been shown to be a good match for the IoT.

At the paradigm level, aggregate programming promotes the specification (construction, reasoning, programming) of global-level computational behaviour, where message exchange across individuals is essentially abstracted away through a declarative style to express interactions. At the modelling level, the field calculus can be leveraged, which expresses computations as transformations of *computational fields* (or *fields* for short), namely, space–time distributed data structures mapping computational events (occurring at a given position of space and time) to computational values (the results of computation at that event). As an example, a set of temperature sensors spread over a building forms a field of temperature values (a field of reals), and a monitor alerting areas where the temperature was above a threshold for the last 10 min is a function from the temperature field to a field of booleans.

We see aggregate computing also as suitable for heterogeneous architectures such as edge computing, where both physical and logical neighbourhood relationships may hold between the terminal (IoT) devices and edge devices, and within or between the other network tiers beyond the edge up to a potentially centralised cloud instance. Its features naturally address challenges that are particular to such contexts (Taherizadeh et al., 2018): mobility of devices, decentralisation and fault tolerance, and interoperability and heterogeneity.

First, aggregate computing copes by design with mobility: On the one hand, it is agnostic to changing network performance and changing neighbours; On the other hand, it can easily be made aware of such notion by accessing sensors on performance data or including timestamps and clock-synchronisation explicitly in all or some messages. Second, aggregate computing is also by nature decentralised: there is no central authority or aggregator per se, and any aggregation requires explicit design on top of broadcast-based communication: the absence of point-to-point connections and an abstraction from actual message transmission leads to a fault tolerant design by construction, as missing or delayed messages or network partitioning are indistinguishable from normal operation. Finally, interoperability and heterogeneity across platforms is – as in many software systems – solved by abstracting the communication layer.

The convenience of aggregate computing for distributed monitoring lies then in its compositionality mechanism, that is fully functional. Hence it induces a compositional translation (the translation of a phrase is determined by combining the translations of its subphrases) of properties expressed in a given logic into a fully-distributed field calculus program whose execution is actually a distributed monitor for such property. Most specifically, phrases are replicated across all nodes, hence an instance of the property is checked by every node of the system altogether.

In this paper we focus on past-CTL logic, a logic able to express properties concerning the knowable past of an event. Such properties include the eventual behaviour of participants in the network, interpreted based on which information has been gathered into an event from the past, and this is shown, through a variety of examples, to give forms of predictive capability. We show that the derived monitors match the optimal efficiency of previous results for past-LTL in a non-distributed setting (Havelund and Rosu, 2002). Hence, this translation showcases how the field calculus can support distributed verification in a high-level compositional way, and paves the way for integration of future extensions of the logic.

The proposed approach aims to target highly-distributed systems where cloud-like support is sporadic or not available, or cannot be successfully exploited, making the monitoring of formulas with future modalities infeasible. In such contexts, using field calculus programming (which comes with fully implemented tools such as FCPP Audrito, 2020 http://fcpp.github.io, a C++ internal DSL and simulator, Protelis Pianini et al., 2015 http://protelis.github.io, a Java external DSL, ScaFi Casadei et al., 2020 https://scafi.github.io, a Scala internal DSL), in combination with past-CTL-based distributed runtime verification as proposed in this paper, can still allow to reasonably monitor interesting properties of distributed systems, as we show by examples and simulations. The main contribution of the present paper is twofold:

- The introduction of past-CTL, a modal logic for expressing temporal properties of a distributed system, inspired by the well-known temporal logics CTL and past-LTL.
- The proposal of a formal, detailed method to automatically translate formulas of past-CTL into fully distributed monitoring systems expressed by aggregate computing, which can be practically deployed and executed.

The remainder of this paper is organised as follows: Section 2 provides the necessary background on field calculus; Section 3 illustrates how the field calculus can be used to implement distributed monitors; Section 4 applies our novel solution to sample problems in aggregate computing; Section 5 discusses related work and Section 6 concludes.

2. A recollection of the field calculus

The *field calculus* (Audrito et al., 2019) is a minimal universal (Audrito et al., 2018a) language to express aggregate computations over distributed networks of (mobile) devices, each asynchronously capable of performing simple local computations and interacting with a neighbourhood by local exchanges of messages. Field calculus provides the necessary mechanism to express and functionally compose such distributed computations, by a level of abstraction that intentionally neglects explicit management of synchronisation, message exchanges between devices, position and quantity of devices, and so on—these aspects are however dealt with "under the hood", namely, at the operational semantics level. In the field calculus, a program P is periodically and asynchronously executed on every device, according to a cyclic schedule executed by each device δ with period T_δ:

1. the device perceives contextual information formed by data provided by sensors, local information stored in the previous round, and messages collected from neighbours since the previous round (older messages get discarded

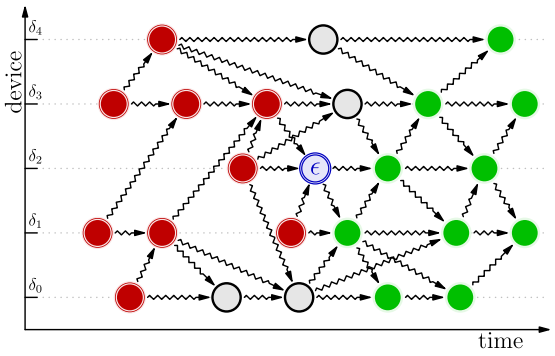

Fig. 1. Sample event structure split into events in the causal past of ϵ ($\epsilon' < \epsilon$, circled solid red), events in the causal future ($\epsilon < \epsilon'$, solid green) and concurrent (non-ordered, hollow black). Events are sorted by the device on which they occur. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

after a certain timeout), the latter in the form of a *neighbouring value* ϕ—essentially a map from neighbour device identifiers δ to values ℓ;
2. when a round starts, the device evaluates the program P considering as input the contextual information gathered as described above (as formalised by the operational semantics);
3. the result of local computation is a data structure that is stored locally, is broadcast to neighbours, and produces output values (possibly fed to actuators).

By repetitive execution of such computation rounds, across space (where devices are located) and time (when devices execute), a global behaviour emerges (Viroli et al., 2018) at the overall network-level of interconnected devices, modelled as a single aggregate machine equipped with a neighbouring-based topology relation. Typically, the neighbouring relation reflects spatial proximity, but it could also be a logical relationship, e.g., connecting master devices to slave devices independently of their position, and the like. The data abstraction manipulated by the "aggregate computing machine" is hence a whole distributed *space–time value* Φ, which maps individual computation *events* ϵ (space–time points where and when a device sends messages at the end of a computation round) to data values v. The set of events, constituting the domain of fields, is then structured according to a *neighbouring* notion, dictating when an event can influence (by message-passing) another.

2.1. Event structures

The causal relationship between events may then be formalised by the classical notion of *event structure* (Lamport, 1978), which we will use in the following sections to interpret temporal logic formulas about distributed computations.

Definition 2.1 (*Event Structure*)**.** An *event structure* $\mathbf{E} = \langle E, \rightsquigarrow, < \rangle$ is a countable 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 $\{\epsilon' \in E \mid \epsilon' < \epsilon\}$ is finite for each ϵ (i.e., $<$ is *locally finite*).[1]

[1] The definition of event structure is usually given just in terms of the causality relation. We have also included the neighbouring relation since it is able to capture message passing details, which are needed to interpret most distributed programs.

Note that the transitive closure condition on $\rightsquigarrow$ also implies that $\rightsquigarrow$ is asymmetric and irreflexive. Fig. 1 provides a sample event structure showing how, given an event ϵ, these relations partition events into "causal past" (events from which information can potentially be carried to ϵ in a message), "causal future" (those to which information from ϵ can be carried) and "concurrent" (events informationally isolated from ϵ) subsets.

Since $<$ is uniquely induced by $\rightsquigarrow$, we omit it whenever convenient, or use its weak form $\leq$.[2] Notice that since $<$ is required to be irreflexive, $\rightsquigarrow$ has to be an acyclic relation, thus inducing a directed acyclic graph (DAG) structure on E. In fact, **E** can be thought of as a DAG with a "neighbouring" relation (modelling message passing) and a "reachability" relation (modelling causal dependence).

Event structures completely abstract away from the information about which device or devices might be performing an actual computation at each event, focussing on which data may be available at every computational step, no matter on what device the computation may be happening. Thus a series of computations on the same device (whether it is fixed or mobile) are to be modelled by a sequence of events $\epsilon_1, \ldots, \epsilon_n$ such that $\epsilon_i \rightsquigarrow \epsilon_{i+1}$, in which message passing is implemented simply by keeping data available on the device for subsequent computations.

We do not make any assumptions on how a particular event structure come into being: each device may for example have changed its spatial position through actuators or external factors (which may have been reflected in particular position- or accelerator-sensor readings on its timeline), derived values may have evolved over time due to computation and communication, which could, but does not have to, have been implemented through field calculus programs.

Any sequence of computation events and message exchanges between them can be represented as an event structure, however, not all event structures are physically realisable by a distributed system following the computational model described at the beginning of this section. The subset of realisable event structures is characterised by the following definition.

Definition 2.2 (*LUIC Augmented Event Structure*)**.** An *augmented event structure* is a tuple $\mathbb{E} = \langle E, \rightsquigarrow, <, d \rangle$ such that $\langle E, \rightsquigarrow, < \rangle$ is an event structure and $d : E \to D$ is a mapping from events to the devices where they happened. We define:

- $\text{next} : E \nrightarrow E$ as the partial function[3] mapping an event ϵ to the unique event $\text{next}(\epsilon)$ such that $\epsilon \rightsquigarrow \text{next}(\epsilon)$ and $d(\epsilon) = d(\text{next}(\epsilon))$, if such an event exists and is unique (i.e., $\text{next}(\epsilon)$ is the computation performed immediately after ϵ on the same device $d(\epsilon)$); and
- $\dashrightarrow \subseteq E \times E$ as the relation such that $\epsilon \dashrightarrow \epsilon'$ (ϵ *implicitly precedes* ϵ') if and only if $\epsilon' \rightsquigarrow \text{next}(\epsilon)$ and $\epsilon' \not\rightsquigarrow \epsilon$.

We say that $\mathbb{E}$ is a *LUIC augmented event structure* if the following coherence constraints are satisfied:

- **Linearity:** if $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ and $d(\epsilon) = d(\epsilon_1) = d(\epsilon_2)$, then $\epsilon_1 = \epsilon_2 = \text{next}(\epsilon)$ (i.e., every event ϵ is a neighbour of at most another one on the same device);
- **Uniqueness:** if $\epsilon_i \rightsquigarrow \epsilon$ for $i = 1, 2$ and $d(\epsilon_1) = d(\epsilon_2)$, then $\epsilon_1 = \epsilon_2$ (i.e., neighbours of an event all happened on different devices);
- **Impersistence:** if $\epsilon \rightsquigarrow \epsilon_i$ for $i = 1, 2$ and $d(\epsilon_1) = d(\epsilon_2) = \delta$, then either $\epsilon_2 = \text{next}^n(\epsilon_1)$ and $\epsilon \rightsquigarrow \text{next}^k(\epsilon_1)$ for all $k \leq n$, or the same happens swapping ϵ_1 with ϵ_2 (i.e., an event reaches a contiguous set of events on a same device);

[2] The weak form of a partial order is defined as $x \leq y$ iff $x < y$ or $x = y$.

[3] With $A \nrightarrow B$ we denote the space of partial functions from A into B.

- **Computation immediacy:** the relation $\rightsquigarrow \cup \dashrightarrow$ is acyclic on E (i.e., explicit causal dependencies $<$ are consistent with implicit time dependencies $\dashrightarrow$).

The first two constraints are necessary for defining the semantics of an aggregate program (denotational semantics in Audrito et al., 2019; Viroli et al., 2019). The third reflects that messages are not retrieved after they are first dropped (and in particular, they are all dropped on device reboots). The last constraint reflects the assumption that computation and communication are modelled as happening instantaneously. In this scenario, the explicit causal dependencies imply additional time dependencies $\epsilon \dashrightarrow \epsilon'$: if ϵ' was able to reach next(ϵ) but not ϵ, the firing of ϵ' must have happened *after* ϵ.

The event structure in Fig. 1 satisfies the LUIC constraints with the represented device assignment. Interpreting this structure in terms of physical devices and message passing, a physical device is instantiated as a chain of events connected by $\rightsquigarrow$ relations (representing evolution of state over time with the device carrying state from one event to the next), and any $\rightsquigarrow$ relation between devices represents information exchange from the tail neighbour to the head neighbour. As a matter of fact, this is a very flexible and permissive model: there are no assumptions about synchronisation, shared identifiers or clocks, or even regularity of events (though of course these things are not prohibited either).

Though rather abstract, the notion of (augmented) event structure is well-suited to ground a semantics for *space–time computations*, intended as "elaborations of distributed data in a network of related events": the causality ordering of events abstracts time, while the presence of concurrent events abstract spatial dislocation. We refer to Audrito et al. (2019) for the definition of a denotational semantics of the field calculus based on augmented event structures and for a formal account of its relation with the operational semantics of the field calculus presented below.

Definition 2.3 (*Space–Time Values*)**.** Let $\mathbf{V}$ be a denumerable universe of allowed computational values and $\mathbb{E}$ be a given LUIC augmented event structure. A *space–time value* Φ in $\mathbb{E}$ is an annotation of the graph $\mathbb{E}$ with labels in $\mathbf{V}$, that is, a tuple $\Phi = \langle \mathbb{E}, f \rangle$ with $f : E \to \mathbf{V}$, taking E as the set of events in $\mathbb{E}$.

Space–time values model data *spatially* distributed across devices and *temporally* distributed across time: in this way, time-evolving inputs, sensor information (e.g., time clock, temperature) and intermediate results of computations (which are naturally time-dependent) are easily represented, attaining maximal generality while ensuring composability of behaviour. These quantities can be manipulated by distributed computations (i.e., consumed as inputs) and can also be created by them (i.e., produced as outputs). Thus, an aggregate computer is a "collective" device manipulating such space–time values, modelled as a *space–time function*.

Definition 2.4 (*Space–Time Function*)**.** Let $\mathbf{V}(\mathbb{E}) = \{\langle \mathbb{E}, f \rangle \mid f : E \to \mathbf{V}\}$ be the set of all possible space–time values in a augmented event structure $\mathbb{E}$. Then, an *n-ary space–time function in* $\mathbb{E}$ is a partial map $\mathtt{f} : \mathbf{V}(\mathbb{E})^n \nrightarrow \mathbf{V}(\mathbb{E})$.

The definition of a space–time function `f` requires every input and output space–time value to exist in the same augmented event structure $\mathbb{E}$. However, it does not specify how the output space–time values are obtained from the inputs, and in fact not all space–time functions `f` are physically realisable by a program, as `f` may violate either causality or Turing-computability (see Audrito et al., 2018a for further details).

The specification of a space–time function can be either done at a low-level (i.e. through local interactions), in order to define programming language constructs and general-purpose building blocks of reusable behaviour, or at a high-level (i.e. by composition of other space–time functions with a global interpretation) in order to design collective adaptive services and whole distributed applications—which ultimately work by getting input fields from sensors and process them to produce output fields to actuators. However, in aggregate computing a distributed program `P` always has both the *local* and *global* interpretations, dually linked: the global interpretation as a space–time function (obtained through a denotational semantics Audrito et al., 2019; Viroli et al., 2019), and the local interpretation as a procedure performed in a firing (defined by the operational semantics presented below).

2.2. Syntax and operational semantics

The syntax of the fragment of the field calculus that fits the needs of this paper is given in Fig. 2—we refer to Audrito et al. (2019, 2020) for a complete presentation of the syntax, type system and operational semantics of field calculus: in the following, we present only what is strictly needed for the remainder of this paper. The overbar notation $\overline{\mathtt{e}}$ is a shorthand for sequences of elements, and multiple overbars are intended to be expanded together, e.g., $\overline{\mathtt{e}}$ stands for the sequence of expressions $\mathtt{e}_1, \ldots, \mathtt{e}_n$ and $\overline{\delta} \mapsto \overline{\ell}$ stand for the map $\delta_1 \mapsto \ell_1, \ldots, \delta_n \mapsto \ell_n$ (that associates device identifiers δ to local values ℓ). The keyword `share` is the main peculiar construct of the field calculus, responsible of both interaction and field dynamics; while `def` and `if` correspond to the standard function definition and the branching expression constructs.

A program `P` consists of a list of function definitions $\overline{\mathtt{F}}$, each written as "`def d(`$\mathtt{x}_1, \ldots, \mathtt{x}_n$`) {e}`", followed by a main expression `e` that is the one executed at each computation round (as well as the one representing the overall field computation, in the global viewpoint). An expression `e` can be:

- A *variable* `x`, used e.g. as formal parameter of functions.
- A *value* `v`, which can be of the following two kinds:
  - A *local value* ℓ, with structure `c(`$\overline{\ell}$`)` or simply `c` when $\overline{\ell}$ is empty (defined via data constructor `c` and arguments $\overline{\ell}$), can be, e.g., a Boolean (`true` or `false`), a number, a string, or a structured value (e.g., a pair `pair(true,5)`).
  - A *neighbouring (field) value* ϕ that associates neighbour device identifiers δ to local values ℓ, e.g., it could be the neighbouring value of distances to neighbours—note that neighbouring field values are not part of the surface syntax, they are produced at runtime by evaluating expressions, as described below.
- A function call `f(`$\overline{\mathtt{e}}$`)`, where `f` can be of two kinds: a *user-declared function* `d` (declared by the keyword `def`, as illustrated above) or a *built-in function* `b`, such as a mathematical or logical operator, a data structure operation, or a function returning the value of a sensor.
- A branching expression `if(`$\mathtt{e}_1$`){`$\mathtt{e}_2$`}{`$\mathtt{e}_3$`}`, used to split a field computation in two isolated sub-networks, where/when $\mathtt{e}_1$ evaluates to `true` or `false`: the result is computation of $\mathtt{e}_2$ in the former area, and $\mathtt{e}_3$ in the latter.
- A share expression `e` $=$ `share(`$\mathtt{e}_1$`){(x)=>`$\mathtt{e}_2$`}`, which incorporates message passing and local state evolution. The result of such expression is obtained by:
  - Gathering the results $\overline{\ell}$ obtained by neighbours $\overline{\delta}$ for the whole expression `e` in their last rounds into a neighbouring field value $\phi = \overline{\delta} \mapsto \overline{\ell}$.

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x | f($\overline{\texttt{e}}$) | v | if(e){e}{e} | share(e){(x)=>e}	expression
f	::=	d | b	function name
v	::=	ℓ | ϕ	value
ℓ	::=	c($\overline{\ell}$)	local value
ϕ	::=	$\overline{\delta} \mapsto \overline{\ell}$	neighbouring field value

Fig. 2. Syntax of the field calculus language.

- Such ϕ may also contain a value for the current device $\delta = \delta_i$ for some i (if it is not the first round it executes e). If not, the result ℓ of evaluating $\texttt{e}_1$ is used as value for the current device δ and incorporated into $\phi := \overline{\delta} \mapsto \overline{\ell}, \delta \mapsto \ell$.
- Expression $\texttt{e}_2$ is then evaluated by substituting ϕ to x, obtaining the overall value ℓ' for e.
- Value ℓ' is broadcast to neighbours, allowing them to use it in constructing their following neighbours' observation ϕ as described above.

This construct is designed for structuring device interaction, but can also be used for evolving a state locally, if the variable x appears in $\texttt{e}_2$ within the built-in operator locHood(ϕ), which extracts the value $\phi(\delta)$ relative to the current device δ from a neighbouring field value ϕ. Note that the evaluation by a device δ of a share-expression within a branch of some if($\texttt{e}_1$) expression, is affected only by the neighbours of δ that, during their last computation cycle, evaluated the same value for the guard $\texttt{e}_1$, since these are the only ones that computed that share expression thus sharing a value for it with δ.

Example 2.5. As an example illustrating the constructs of field calculus, first consider the expression:

```
share (false) { (old) => f || anyHood(old) }
```

where f is some captured variable, and anyHood is a built-in operator collapsing a field $\phi = \overline{\delta} \mapsto \overline{b}$ of Boolean values $\overline{b}$ into the disjunction of its constituent values $\bigvee \overline{b}$. When this expression is evaluated for the first time, no messages are available from neighbours, hence the gathered ϕ is comprised only of the value of $\texttt{e}_1$ for the current device: $\phi = \delta \mapsto$ false. As ϕ is substituted for old, we compute anyHood(ϕ) which is false, and finally f || false which is f. When this same expression if evaluated later on, ϕ may contain messages from neighbours. If any one of them is true, anyHood(ϕ) will be true and so will the whole share expression. If none of them is true, the value of the whole expression will be again equal to f. This distributed behaviour can be understood as computing whether f has *ever* been true by *gossipping* whether this is the case across neighbours. As such, this expression can be conveniently encapsulated into the following function definition:

```
def gossip-ever(f) { share (false) { (old) => f
    || anyHood(old) } }
```

As discussed above, function gossip-ever takes a boolean field f and whenever it holds a true value in a device, this gets propagated throughout the network by gossipping.

The built-in functions and data constructors used in this paper are listed in Fig. 3 together with their types and formal interpretations—all of these operators are natively available in existing implementations of the field calculus, including FCPP (used in the case studies presented in Section 4). In this paper, we use types bool (Boolean values), field[bool] for neighbouring fields built from values of type bool, pair[bool, bool] for pairs of Boolean values, and types $(\overline{T}) \to T$ for functions. In the remainder of this paper, we also use $[\texttt{e}_1, \texttt{e}_2]$ as short for $\texttt{pair}(\texttt{e}_1, \texttt{e}_2)$.

3. Monitoring of past-CTL properties in field calculus

In this section, we introduce a temporal logic (past-CTL) designed for reasoning about properties of (augmented) event structures, which can be translated in field calculus in order to allow efficient recursive computation of the truth value of such formulas, assuming that every participant is evaluating the same property from their perspective with regard to any quantifiers. This logic will be based on atomic propositions (observables in a particular state), which we assume to be backed by implicit Boolean space–time values. This will allow us not to consider a mix of temporal operators and (boolean) field calculus expressions, but rather completely encapsulate the latter in the abstract propositions.

Section 3.1 introduces the syntax of past-CTL, while Section 3.2 provides its formal interpretation on an augmented event structure. Section 3.3 introduces a translation of past-CTL into field calculus, coherently with the formal interpretation, and Section 3.4 concludes discussing the applicability of past-CTL and its limited future prediction capabilities.

3.1. Past-CTL syntax

Fig. 4 presents the past-CTL syntax, following the syntax of past-LTL operators in literature (see e.g. Gigante et al., 2017) and extending them as CTL extends LTL. This logic is almost identical to traditional CTL, with two main differences:

- temporal operators are interpreted in the past (and thus their names are changed accordingly), along paths of message exchanges in an event structure, composed of events that all happened in the past (and are not alternative realities);
- there are un-quantified versions of the operators along with quantified versions, which refer to the linear past on a same device (and thus behave as past-LTL operators).

Formulas in past-CTL can be interpreted in augmented event structures (see Section 2.1), giving a definite truth value for each event (and thus producing a space–time value), as we shall see in Section 3.2. The operators have the following informal meaning:

- similarly to CTL, path quantifiers A and E refer to the fact that the corresponding formula holds in every (resp. some) path of messages, starting from an initial event of a device

Constructors:			
`true, false`	=	$() \to$ `bool`	$\top,\ \bot$
Built-ins:			
`!`	=	$($`bool`$) \to$ `bool`	$\neg$
`||, &&`	=	$($`bool, bool`$) \to$ `bool`	$\vee,\ \wedge$
`<=, ==`	=	$($`bool, bool`$) \to$ `bool`	$\leq,\ =$
`pair`	=	$($`bool, bool`$) \to$ `pair[bool, bool]`	$(p, q) \mapsto [p, q]$
`fst, snd`	=	`pair[bool, bool]` $\to$ `bool`	$[p, q] \mapsto p$ (resp. q)
`locHood`	=	$($`field[bool]`$) \to$ `bool`	$\phi \mapsto \phi(\delta)$
`anyHood`	=	$($`field[bool]`$) \to$ `bool`	$\phi \mapsto \bigvee \{\phi(\delta') \mid \delta' \in \mathbf{dom}(\phi)\}$
`allHood`	=	$($`field[bool]`$) \to$ `bool`	$\phi \mapsto \bigwedge \{\phi(\delta') \mid \delta' \in \mathbf{dom}(\phi)\}$

Fig. 3. Types and interpretations of the data constructors and built-in functions used throughout this paper. We use δ to denote the current device of the current event, and notation $\mathbf{dom}(\phi)$ to denote the domain $\overline{\delta}$ of a function $\phi = \overline{\delta} \mapsto \overline{\ell}$.

$$\begin{aligned}\psi \ ::= \ & \bot \mid \top \mid q \mid (\neg\psi) \mid (\psi \wedge \psi) \mid (\psi \vee \psi) \mid (\psi \Rightarrow \psi) \mid (\psi \Leftrightarrow \psi) && \text{logical op.}\\ & \mid (\mathrm{Y}\,\psi) \mid (\mathrm{AY}\,\psi) \mid (\mathrm{EY}\,\psi) \mid (\psi\,\mathrm{S}\,\psi) \mid (\psi\,\mathrm{AS}\,\psi) \mid (\psi\,\mathrm{ES}\,\psi) && \text{temporal op.}\\ & \mid (\mathrm{P}\,\psi) \mid (\mathrm{AP}\,\psi) \mid (\mathrm{EP}\,\psi) \mid (\mathrm{H}\,\psi) \mid (\mathrm{AH}\,\psi) \mid (\mathrm{EH}\,\psi)\end{aligned}$$

Fig. 4. Syntax of past-CTL.

and ending in the current event; whereas the absence of path quantifiers refers to the fact that the corresponding formula holds in the special path of messages connecting the events happened in the current device;

- operator Y *"yesterday"* postulates that ψ held in the previous event on the considered path (on the same device for Y, on every other device for AY, in another device for EY);
- operator S *"since"* postulates that its second argument held in some past event in the considered path, and its first argument has held since then;
- operator P *"previously"* postulates that ψ held in some past event on the considered path;
- operator H *"historically"* postulates that ψ held in every past event on the considered path.

The list of operators presented above is redundant. A minimal set of operators is $\neg, \vee$, Y, AY, S, AS, ES, the others being expressible as:

- $\mathrm{EY}\,\psi := \neg\,\mathrm{AY}\,\neg\psi$;
- $\mathrm{P}\,\psi := \top\,\mathrm{S}\,\psi$ (similarly for AP, EP with AS, ES);
- $\mathrm{H}\,\psi := \neg\,\mathrm{P}\,\neg\psi$ (similarly for AH, EH with EP, AP).

Example 3.1. $(\mathrm{AH}(r \Rightarrow (\mathrm{Y}(\neg r\,\mathrm{S}\,q))))$ is a valid past-CTL formula, with the informal meaning "it is always and everywhere the case, that if r holds then in the previous round r was false since a past round where q was true". In the following we shall apply usual operator precedence conventions (assuming temporal operators to bind tighter than logical ones), in order to write the same formula as $\mathrm{AH}(r \Rightarrow \mathrm{Y}(\neg r\,\mathrm{S}\,q))$.

Referring to the scenario on request/response matching over a logical layered network (described in the introduction), we can imagine r to correspond to a *response* and q to correspond to a *request*. In this context, the formula can be interpreted as "every response corresponds to a preceding request": in particular, $r \Rightarrow \mathrm{Y}(\neg r\,\mathrm{S}\,q)$ holds in an event with a recent trace of the kind $r, \neg r, \ldots, \neg r, q$.

3.2. Semantic interpretation on augmented event structures

Given a past-CTL formula ψ, an augmented event structure $\mathbb{E}$, and a Boolean space–time value Φ_q in it for each propositional symbol q occurring in ψ, we can interpret ψ in $\mathbb{E}$ obtaining a space–time value $\Phi = \mathbb{E}[\![\psi]\!]$, such that $\mathbb{E}[\![\psi]\!](\epsilon)$ is defined by causal recursion on ϵ as follows.

- $\mathbb{E}[\![\top]\!](\epsilon) = \top$, and $\mathbb{E}[\![q]\!](\epsilon)$ is $\Phi_q(\epsilon)$;
- $\mathbb{E}[\![\neg\psi]\!](\epsilon) = \neg\mathbb{E}[\![\psi]\!](\epsilon)$ and $\mathbb{E}[\![\psi_1 \vee \psi_2]\!](\epsilon) = \mathbb{E}[\![\psi_1]\!](\epsilon) \vee \mathbb{E}[\![\psi_2]\!](\epsilon)$;
- $\mathbb{E}[\![\mathrm{Y}\,\psi]\!](\epsilon) = \mathbb{E}[\![\psi]\!](\epsilon')$ where ϵ' is the event preceding ϵ on the same device (if it exists, $\mathbb{E}[\![\mathrm{Y}\,\psi]\!](\epsilon) = \bot$ otherwise);
- $\mathbb{E}[\![\mathrm{AY}\,\psi]\!](\epsilon) = \bigwedge_{\epsilon' \rightsquigarrow \epsilon} \mathbb{E}[\![\psi]\!](\epsilon')$ (i.e., ψ is true in each preceding event ϵ');
- $\mathbb{E}[\![\psi_1\,\mathrm{AS}\,\psi_2]\!](\epsilon)$ holds iff for every path $\epsilon_1 \rightsquigarrow \ldots \rightsquigarrow \epsilon_n = \epsilon$ such that ϵ_1 has no neighbours, $\mathbb{E}[\![\psi_2]\!](\epsilon_i)$ holds for some i and $\mathbb{E}[\![\psi_1]\!](\epsilon_j)$ holds for each $j = i+1 \ldots n$;
- $\mathbb{E}[\![\psi_1\,\mathrm{ES}\,\psi_2]\!](\epsilon)$ holds iff it exists a path $\epsilon_1 \rightsquigarrow \ldots \rightsquigarrow \epsilon_n = \epsilon$ such that $\mathbb{E}[\![\psi_2]\!](\epsilon_1)$ holds and $\mathbb{E}[\![\psi_1]\!](\epsilon_i)$ holds for $i = 2 \ldots n$;
- $\mathbb{E}[\![\psi_1\,\mathrm{S}\,\psi_2]\!](\epsilon)$ holds iff it exists a path $\epsilon_1 \rightsquigarrow \ldots \rightsquigarrow \epsilon_n = \epsilon$ of events all occurring on the same device $\delta = d(\epsilon)$,[4] such that $\mathbb{E}[\![\psi_2]\!](\epsilon_1)$ holds and $\mathbb{E}[\![\psi_1]\!](\epsilon_i)$ holds for $i = 2 \ldots n$.

Semantics of derived operators can be inferred accordingly.

Example 3.2. The semantics of formula $\psi := \mathrm{AH}(r \Rightarrow \mathrm{Y}(\neg r\,\mathrm{S}\,q))$ is:

$$\begin{aligned}\mathbb{E}[\![\psi]\!](\epsilon) &= \bigwedge_{\epsilon' \leq \epsilon} \mathbb{E}[\![r \Rightarrow \mathrm{Y}(\neg r\,\mathrm{S}\,q)]\!](\epsilon')\\ &= \bigwedge_{\epsilon' \leq \epsilon} \left[\Phi_r(\epsilon') \Rightarrow \mathbb{E}[\![\mathrm{Y}(\neg r\,\mathrm{S}\,q)]\!](\epsilon')\right]\\ &= \bigwedge_{\epsilon' \leq \epsilon} \left[\Phi_r(\epsilon') \Rightarrow \bigvee_{\substack{\epsilon'' \rightsquigarrow \epsilon'\\ d(\epsilon'') = d(\epsilon')}} \mathbb{E}[\![\neg r\,\mathrm{S}\,q]\!](\epsilon'')\right]\\ &= \bigwedge_{\epsilon' \leq \epsilon} \left[\Phi_r(\epsilon') \Rightarrow \bigvee_{\substack{\epsilon'' \rightsquigarrow \epsilon'\\ d(\epsilon'') = d(\epsilon')}} \bigvee_{\substack{\epsilon_1 \rightsquigarrow \epsilon_2 \ldots \rightsquigarrow \epsilon_n = \epsilon''\\ d(\epsilon_i) = d(\epsilon_j)\ \forall i,j}} \left(\Phi_q(\epsilon_1) \wedge \bigwedge_{i \geq 2} \neg\Phi_r(\epsilon_i)\right)\right]\end{aligned}$$

[4] We recall that $d(\epsilon)$, according to the definition of augmented event structure, corresponds to the device where ϵ happened.

```
def  Y(f) { snd(share([false,false]){(old) => [f, locHood(fst(old))]}) }
def AY(f) { snd(share([true, true ]){(old) => [f, allHood(fst(old))]}) }
def EY(f) { snd(share([false,false]){(old) => [f, anyHood(fst(old))]}) }
def  S(f1, f2) { share (false) {(old) => f2 || (f1 && locHood(old))} }
def AS(f1, f2) { share (false) {(old) => f2 || (f1 && allHood(old))} }
def ES(f1, f2) { share (false) {(old) => f2 || (f1 && anyHood(old))} }
def  P(f) { share (false) {(old) => f || locHood(old)} }
def AP(f) { share (false) {(old) => f || allHood(old)} }
def EP(f) { share (false) {(old) => f || anyHood(old)} }
def  H(f) { share (true) {(old) => f && locHood(old)} }
def AH(f) { share (true) {(old) => f && allHood(old)} }
def EH(f) { share (true) {(old) => f && anyHood(old)} }
```

Fig. 5. Encoding of past-CTL operators as field calculus functions.

$[\![\top]\!]$	=	`true`	$[\![\psi_1 \vee \psi_2]\!]$	=	$[\![\psi_1]\!]$ `||` $[\![\psi_2]\!]$
$[\![\bot]\!]$	=	`false`	$[\![\psi_1 \wedge \psi_2]\!]$	=	$[\![\psi_1]\!]$ `&&` $[\![\psi_2]\!]$
$[\![q]\!]$	=	`q()`	$[\![\psi_1 \Rightarrow \psi_2]\!]$	=	$[\![\psi_1]\!]$ `<=` $[\![\psi_2]\!]$
$[\![\neg\psi]\!]$	=	`!`$[\![\psi]\!]$	$[\![\psi_1 \Leftrightarrow \psi_2]\!]$	=	$[\![\psi_1]\!]$ `==` $[\![\psi_2]\!]$
$[\![\mathrm{Y}\,\psi]\!]$	=	`Y(`$[\![\psi]\!]$`)`	$[\![\psi_1 \mathrm{S}\,\psi_2]\!]$	=	`S(`$[\![\psi_1]\!]$`,` $[\![\psi_2]\!]$`)`
$[\![\mathrm{AY}\,\psi]\!]$	=	`AY(`$[\![\psi]\!]$`)`	$[\![\psi_1 \mathrm{AS}\,\psi_2]\!]$	=	`AS(`$[\![\psi_1]\!]$`,` $[\![\psi_2]\!]$`)`
$[\![\mathrm{EY}\,\psi]\!]$	=	`EY(`$[\![\psi]\!]$`)`	$[\![\psi_1 \mathrm{ES}\,\psi_2]\!]$	=	`ES(`$[\![\psi_1]\!]$`,` $[\![\psi_2]\!]$`)`
$[\![\mathrm{P}\,\psi]\!]$	=	`P(`$[\![\psi]\!]$`)`	$[\![\mathrm{H}\,\psi]\!]$	=	`H(`$[\![\psi]\!]$`)`
$[\![\mathrm{AP}\,\psi]\!]$	=	`AP(`$[\![\psi]\!]$`)`	$[\![\mathrm{AH}\,\psi]\!]$	=	`AH(`$[\![\psi]\!]$`)`
$[\![\mathrm{EP}\,\psi]\!]$	=	`EP(`$[\![\psi]\!]$`)`	$[\![\mathrm{EH}\,\psi]\!]$	=	`EH(`$[\![\psi]\!]$`)`

Fig. 6. Straightforward compositional translation of past-CTL into field calculus.

3.3. Automatic translation in field calculus

Past-CTL operators can be encoded as field calculus functions, taking the Boolean value of their sub-formulas as arguments. Fig. 5 shows a possible implementation for them, assuming that:

- `fst`, `snd` are operators extracting the first and second element of a pair;
- `anyHood` is as in Example 2.5;
- `allHood` is a built-in operator collapsing a field ϕ of Boolean values into the conjunction of its constituent values $\bigwedge \phi$;
- `locHood` is a built-in operator collapsing a field ϕ into the value $\phi(\delta)$ held by ϕ for the current device δ.

Using these functions, past-CTL formulas can be straightforwardly translated into field calculus, as shown in Fig. 6. We translate atomic propositions q into built-in function calls getting their value from some external environment. We translate logical operators into their corresponding built-ins, assuming that `false < true`, as in common programming languages such as C/C++ or Python. Temporal operators are translated by calling the corresponding functions.

Example 3.3. The translation of formula $\mathrm{AH}(r \Rightarrow \mathrm{Y}(\neg r\,\mathrm{S}\,q))$ is `AH(r <= Y(S(!r, q)))`.

We recall that the field calculus programs obtained from past-CTL formulas can be straightforwardly encoded in one of the DSLs (FCPP, Protelis and ScaFi, mentioned in Section 1) that implement its programming model and then deployed and executed on a network of devices.

It is worth observing that the translation is fully compositional, and resembles the syntax of a non-distributed monitor, even though it is executed in a fully distributed manner thanks to the peculiar field calculus execution model and semantics. Compositionality also allows to easily handle extensions of the logic, as additional operators would just translate into additional rows in the translation table. Furthermore, computational complexity matches that of the best known monitors for past-LTL formulas in a non-distributed setting (Havelund and Rosu, 2002).

Theorem 3.4 (*From past-CTL to Field Calculus*)**.** *The translation in Fig. 6 is correct, computes in space/time which is linear in the size of the formula (in each event), and exchanges 1 bit for every* S, P, H *operator and 2 bits for every* Y *operator.*[5]

Proof. Let ψ be composed from sub-formulas $\psi_1, \ldots, \psi_n$ ($0 \leq n \leq 2$) and proceed by structural induction on formulas. The translation of ψ consists of a function call, where the body computes in constant space/time from the argument values, that are obtained from the translations of ψ_i which compute in linear space/time by inductive hypothesis. It follows that the overall space/time required is also linear. Furthermore, if ψ is not built from a temporal operator, its direct translation does not involve message-exchanging constructs. If it is built from a S, P, H operator, it consists of a single message-exchanging construct (here: `share`) sharing a single Boolean value (1 bit). If it is built from a Y operator, it also consists of a single message-exchanging construct, which however shares a pair of two Boolean values (2 bits).

If ψ is built from a traditional logic operator, correctness is trivial. If $\psi = \mathrm{Y}\,\psi_1$, the translation of ψ calls a function, which takes the second element of a pair consisting of the current value $\mathbb{E}[\![\psi_1]\!](\epsilon)$ of ψ_1 (by inductive hypothesis) together with the value the first element had in the previous event on the same device, that is the previous value of ψ_1 on the same device. If $\psi = \mathrm{AY}\,\psi_1$, the translation of ψ is taking the second element of a pair which consists of $\mathbb{E}[\![\psi_1]\!](\epsilon)$ together with the conjunction of the values $\mathbb{E}[\![\psi_1]\!](\epsilon')$ the first element had in previous events. A similar argument applies for EY.

If $\psi = \psi_1\,\mathrm{S}\,\psi_2$ and $\mathbb{E}[\![\psi_2]\!](\epsilon)$ is true, then $\mathbb{E}[\![\psi]\!](\epsilon)$ is true and so is the translation. If it is not, then ψ_1 has to be true on previous events on the same device, since a preceding event where ψ_2 held. This is equivalent to asking that ψ_1 holds in the current event, and ψ held on the preceding event, which in field calculus corresponds to `f1 && locHood(old)`. Similar arguments apply for AS, ES. □

[5] Optimisations in order to exchange 1 bit for every Y operator are possible, but require additional field calculus constructs, thus we decided to omit them for presentation simplicity.

Remark 3.5. Theorem 3.4 shows that the field calculus translation provided is asymptotically optimal, since some past-CTL formulas cannot be computed with sub-linear time or space, nor with message size lower than 1 bit per temporal connective. Such a family of formulas is, for example, $\bigwedge_{i \in N} \mathrm{EP}\, q_i$ where q_i are independent atomic propositions. A computation in sub-linear time cannot possibly check every q_i and thus cannot be correct. Furthermore, it is necessary for every event to message whether each q_i has ever become true, otherwise some past information would be lost making it impossible to reconstruct the correct answer.

3.4. Practical use of persistent past-CTL properties

Past-CTL logic states properties about the past of a given event in space and time. Like traditional semantics of CTL, our semantics is defined for a state (event) by checking derived properties on preceding states and combining them into a verdict *for that state*. The decision to check a logical property for one state, some states or all states of an event structure is up to the end-user. However, as we are considering an event structure that is evolving at runtime and hence cannot make this choice explicitly, we re-interpret a formula as stating properties about *what we know so far of the future* of a system: our mechanism implicitly checks a given property on every state, conceptually "restarting" evaluation of the overall (initial) formula from each new state. We say that a formula is persistent in the sense that it is checked again and again, based on potentially new information.

A main concern for runtime verification are safety or liveness properties ("something bad never happens"/"something good eventually happens"), and through our choice of a past-time logics we have cemented our viewpoint into the past, clearly defining on which observations we want to base our evaluation. However, from the perspective of an agent starting up, taking such a property and (only) evaluating it in the initial state (like the root of a Kripke structure) where it does not have a past yet or did not yet receive any information from neighbours clearly does not make sense. Instead, we want to know the verdict for every state that an agent goes through.

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

For instance, assume that we are interested in whether a certain property φ is ever true in any point of space and time. In an event infinitely far in the future, this could be checked through $\mathrm{EP}\,\varphi$. The same formula can be interpreted in a "present" event as computing whether φ has already become true ($\top$ result), or *not yet* ($\bot$ result) based on the limited knowledge so far. Similarly, if we want to check whether a certain assumption φ is violated, we could monitor it through $\mathrm{AH}\,\varphi$, interpreting its result on events as either *good so far* ($\top$) or *violation detected* ($\bot$).

The predictive capabilities of past-CTL formulas are mostly restricted to the interpretation of their outermost connective, and are thus more limited compared to those of a full-fledged future logic, which is sometimes able to perform inferences in order to detect events ahead of time. However, that increase in predictive capability comes with a higher computational cost; and in many practical scenarios the improvement is not large enough (if at all present) to justify the additional costs. In the following section, we will inspect some such examples.

3.5. Integration of field calculus monitors and external applications

Thus far, we have discussed how to express relevant properties of a distributed system in past-CTL and how to generate distributed Field Calculus programs for monitoring such properties. However, when the system to be monitored is itself made of software, or is wrapped in a software system, it becomes fundamental to integrate it with the generated monitors. In this section we briefly consider some lines for realising such integration.

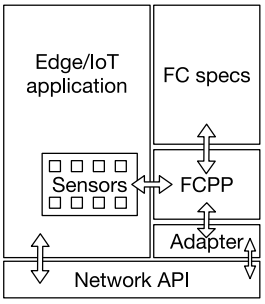

Fig. 7. Architecture/Integration.

Let us first consider a C/C++ distributed application. Fig. 7 gives a high-level view of how formulas in FC are checked by the FCPP runtime in parallel to a running application. The FCPP runtime needs, on one hand, to access to the sensor data wrapped in the application, and on the other hand an adapter to access the network, either in the form of physical access to e.g. Ethernet, 5G, . . ., or a software defined network.

At the source code level, first of all the application must import the FCPP library and integrate the generated formulas (as we have done for the sample applications described in Section 4). Notice that no knowledge on aggregate computing is required: only an informal understanding of temporal logics, so that the programmer can express the formulas he needs to monitor (everything else is fully automated). The network adapter code is provided by implementing a transceiver interface for the underlying platform, i.e. `os::uid` (the unique id of the devices on the network) and `os::transceiver` (functions to send and receive data on the network). Then, the monitor code can be run as a *net* object in its separate thread as in the following.

```
#include "lib/fcpp.hpp"

namespace fcpp {
  namespace os {
    device_t uid() { ... } // unique identifier
    struct transceiver { ... }; // packet send-receive interface
  }

  struct P {}; ... // propositional variables
  struct M {}; ... // monitored formulas

  MAIN() {
    bool p = node.storage(P{}); ... // access propositions
    bool m = ... // compute monitors (past-CTL formulas)
    node.storage(M{}) = m; .. // write monitors' results
  }

  using net_type = component::monitor<
    component::tags::program<main>,
    component::tags::exports<bool>,
    component::tags::tuple_store<P, bool ... M, bool ...>
  >::net;
}

int main() {
        using namespace fcpp;
        net_type network{common::make_tagged_tuple<>()};
        std::thread monitor_thread(network.run);
        ... // main application code
}
```

When the main application code needs to provide sensor data to the monitor, it can access the `storage` of the current monitor node to set the current values of the propositional variables

that act as inputs in the property being monitored, and then release a new round of communication among the monitor nodes. Similarly, to access the current value of the monitored property, it is sufficient to read it from the node `storage`:

```
typename net_type::lock_type l;
auto& node = net.node_at(os::uid(), l);
node.storage(P{}) = ...
node.trigger();
m = node.storage(M{})
... // react to violations of the property
```

The integration of JVM-based applications (e.g., Java, Scala, Kotlin) with the past-CTL monitors is conceptually similar to the one just described for C++, except that the Scafi library should be used in place of FCPP.

Finally, a software system based on any technology can be integrated with FCPP monitors by writing an ad-hoc, small middleware C++ system that, on one hand, wraps the Field Calculus monitors and, on the other hand, exchanges messages with the monitored system exploiting inter-process communication mechanisms such as message queues, pipes, or even files.

4. Sample applications

In this section, we present sample applications of past-CTL logic to the monitoring of distributed systems coming from various settings: service discovery (Section 4.1), swarm control (Section 4.2), disaster management (Section 4.3), and smart home scenarios (Section 4.4). All the simulations have been performed within the FCPP language and simulator (Audrito, 2020) and are available online.[6] In every scenario, we assumed the computation frequency of devices to be about 1Hz (with 10% standard deviation), and plotted the percentage of *false* values for monitored formulas across devices in the network over time. The scenarios involve both static sensors or logical devices, units on humans moving at a walking speed of 1.4 m/s (Levine and Norenzayan, 1999), and drones flying with a top speed of 15 m/s (54 km/h), which is an average limit for commercial UAVs.

4.1. Service discovery

We consider a network with three types of nodes: *cloud* nodes located on a small central circle; *fog* nodes located on an inner circle at some distance from the cloud; and *edge* nodes located on an outer circle further away from the cloud. Proposition rq_i is true iff the current node has just requested a service of type i, and proposition rs_i is true iff the current node has just received a response for its type-i request.

Consider the following *global* properties, where N is the set of possible request types:

1. *timeout*: every request of type i should be followed by a suitable response (i.e., of type i) from some other node within n time instants. In a negative form, we can express it as:

   $$\neg\,\mathrm{EP}(no\_reply(rq_i, rs_i, n))$$

   where *no_reply* is defined as[7]:

   $$no\_reply(rq, rs, n) = \begin{cases} rq & \text{if } n = 0 \\ \neg rs \wedge \mathrm{Y}\, no\_reply(rq, rs, n-1) & \text{otherwise.} \end{cases}$$

   Note that we are actually defining as many properties as the number of request types N.

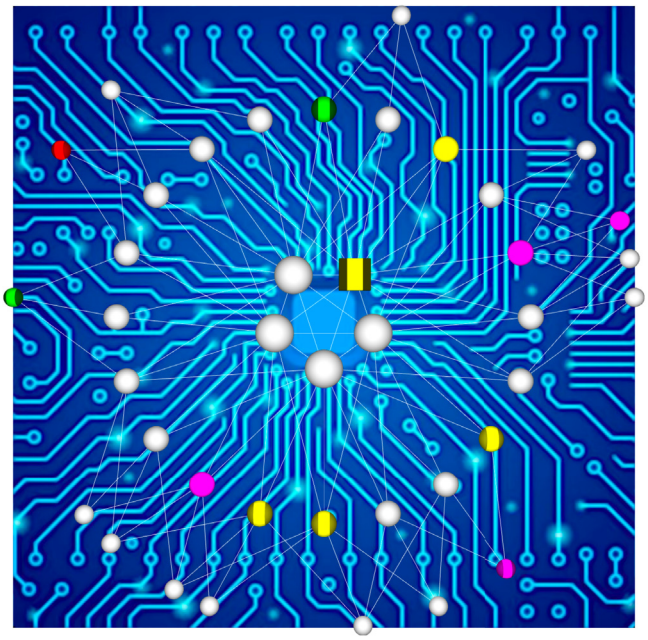

Fig. 8. Screenshot of a simulated network. Coloured nodes have a pending request (colour based on request type, darker sides for older requests), square nodes if the monitor is failed. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

2. *spurious*: a response of any type $i \in N$ should be received only if a corresponding request was issued at some previous time instant. In formulas:

   $$\bigwedge_{i \in N} \mathrm{AH}(rs_i \Rightarrow \mathrm{Y}(\neg rs_i\, \mathrm{S}\, rq_i))$$

   This formula has already been considered in Examples 3.1–3.3. Notice that here we consider a single formula for all the types $i \in N$ (external $\wedge$ ranging on $i \in N$).
3. *double req*: nodes issue requests one at a time, i.e., they wait for a response before issuing another request. Again in a negative form, and considering a single formula for all $i \in N$, we can express it as:

   $$\bigwedge_{i \in N} \neg\,\mathrm{EP}(\mathrm{Y}((\neg rs_i\, \mathrm{S}\, rq_i) \wedge rq_i))$$

We evaluated this scenario in a logical environment, depicted in a 2D plane for presentation purposes (see Fig. 8 for a snapshot), where network nodes are as follows: 5 central cloud nodes, all connected to each other; 20 intermediate fog nodes, each connected with 2 or 3 cloud nodes; and 50 edge nodes each connected with 3 or 4 fog nodes. Note that neither fog nodes, nor edge nodes can communicate with their peers. The network topology of cloud and fog nodes is static, while every edge node appears and disappears at a random time after the beginning of the simulation.

Nodes move back and forth between two states: *computing* (white nodes) and *pending request* (coloured nodes). The transition from *computing* to *pending request* happens when the node sends a request rq_i; the opposite transition happens when the node receives a response rs_i.

The type of the current request is represented by the colour of the node: yellow, green, red, and fuchsia for types 1 to 4 (white nodes do not have a pending request). As the node spends time waiting for a response, its sides become darker. The simulator has different (average) response times for the different request types, with the time for type 1 being the highest, and the others

[6] https://github.com/Harniver/past-ctl-monitoring.

[7] It should be clear that the recursive definition of *no_reply* is easily expressed in past-CTL once n has been fixed.

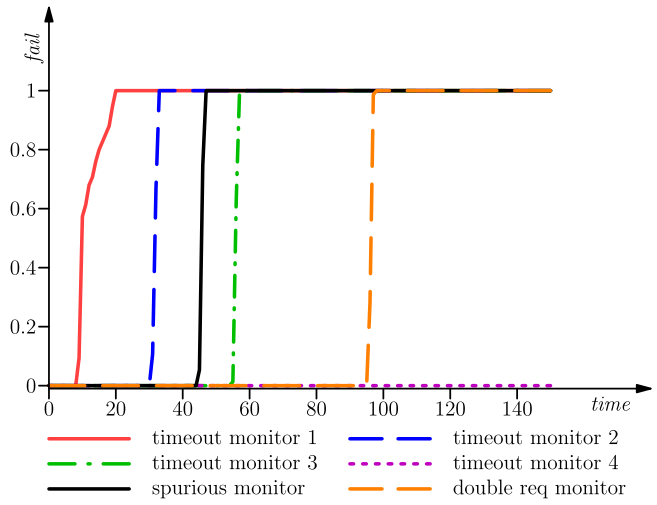

Fig. 9. Monitors evolution in time. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

progressively lower. As expected, the monitors of the *timeout* properties detect a timeout for type 1 earlier than for types 2 and 3, while no timeout for type 4 is detected in the simulation horizon; this is illustrated by Fig. 9, which plots, for each request type, the rate of nodes that know that the *timeout* property has been violated. Note that, in Fig. 8, the shape of nodes that know about a timeout violation changes from sphere to cube.

In order to test the monitoring of the *spurious* property above, we have injected the possibility for a node to receive a reply not matching the type of its last request. In particular, the solid black line in the graph of Fig. 9 shows that the violation of the *spurious* property is quickly propagated to all the nodes after time 40.

Similarly, for testing the monitoring of the *double req* property, we have injected the possibility for a node to occasionally issue a second request before receiving a response. The dashed orange line in the graph of Fig. 9 shows that the violation of the *double req* property is quickly propagated to all the nodes at a time close to 100.

4.2. Drones recognition

Suppose that N areas need to be handled by drones in some way (perform a recognition, spray substances, and similar). Proposition q_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 neatly encode that at the same point in time, different drones may have different views of the global system. Consider the properties:

- Every area gets eventually handled by a drone (true means success, false means "not yet"): $\bigwedge_{i \in N} \mathrm{EP}\, q_i$. Observe the requirements that this property imposes on the underlying event structure: if e.g. we would like to use this property as a criterion for distributed termination, we have to make sure that we implement a system where eventually every drone will somehow see the past of every other drone.
- No drone is handling an area that it knows to be already handled (true means "ok so far", false means failure). In other words, it never happens that a drone is handling an area that it knows to be already handled: $\bigwedge_{i \in N} \mathrm{AH}\, \neg(q_i \wedge \mathrm{EY}(\mathrm{EP}\, q_i))$.

The first property can be understood as *liveness*, while the second as *safety*.

We evaluated this scenario in an open 3D environment 1000 m $\times$ 1000 m $\times$ 100 m large (see Fig. 10 (left) for a snapshot), subdivided into four square areas each with a communication tower at its middle (brown/red cubes). 50 drones swept the area to perform 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. We assumed that drones and towers followed an edge-like connection topology: towers were all connected with each other, and drones were connected to the one or two towers that were close enough to them.

The four towers issued "handling"-requests at a random time between 0 s and 200 s, instructing the closest waiting drones to reach them. Shortly after having issued the request, every tower's request was served and the corresponding handling monitor (liveness) quickly started becoming true in every node in the network (see Fig. 10 (right), showing the evolution in time of the percentage of nodes evaluating the monitors to $\bot$). Due to a fault in the algorithm, a second drone reached and handled tower 2 at $t = 80$, causing the corresponding safety monitor to quickly become false in every device of the network. The safety monitors of the other three areas remained true for the whole simulation time, since no other re-servicing occurred.

Notice that an individual breakdown of the propositions composing the monitor through boolean operators allows for an easier understanding of the precise behaviour of the system. This is particularly true if a proposition is composed from a liveness and safety property together: the conjunction of such properties may be continuously false, and we could not tell apart situations in which liveness has not been reached yet, from situations in which liveness has been reached only while breaking safety requirements.

4.3. Crowd safety

Previous works (Beal et al., 2015) present field-calculus solutions to the problem of safe crowd dispersal for disaster management. Armed with our new formalism, we can now specify the property that every agent does not enter back into the unsafe area after having evacuated from the unsafe area. The all-quantification here is implicit, as we assume that every agent is evaluating the same property, and only considering its own past. For this example, we assume a sensor (or field-calculus function) *safe* in every agent, and an indicator for whether an evacuation *alert* has been given, and the device should suggest to move from an unsafe area to safety. We specify that whenever in safety during an alert, we do not regress. As such, we do not use any branching-time features, and we can specify the property as:

$$\varphi_1 := \mathrm{H}(\mathrm{Y}(\mathit{safe} \wedge \mathit{alert}) \rightarrow (\mathit{safe} \vee \neg \mathit{alert})).$$

It is easy to see how the truth value of this property may evolve. Starting out in an unsafe location, the overall verdict remains *true* even once safety is reached. It is also obvious that the verdict may switch to *false* depending on external factors, such as the agent moving or the unsafe area encroaching the agent.

In a next step, we could then lift that monitor to an observer which specifies that everyone in the past cone of events has reached safety without intermittently getting into danger again. We achieve this by switching out the linear-time past operator H to the branching-operator AH:

$$\varphi_2 := \mathrm{AH}(\mathrm{Y}(\mathit{safe} \wedge \mathit{alert}) \rightarrow (\mathit{safe} \vee \neg \mathit{alert})).$$

Note that the verdict *true* then may become invalidated for agents that learn from/come in contact with agents which themselves observed violations of this property over time.

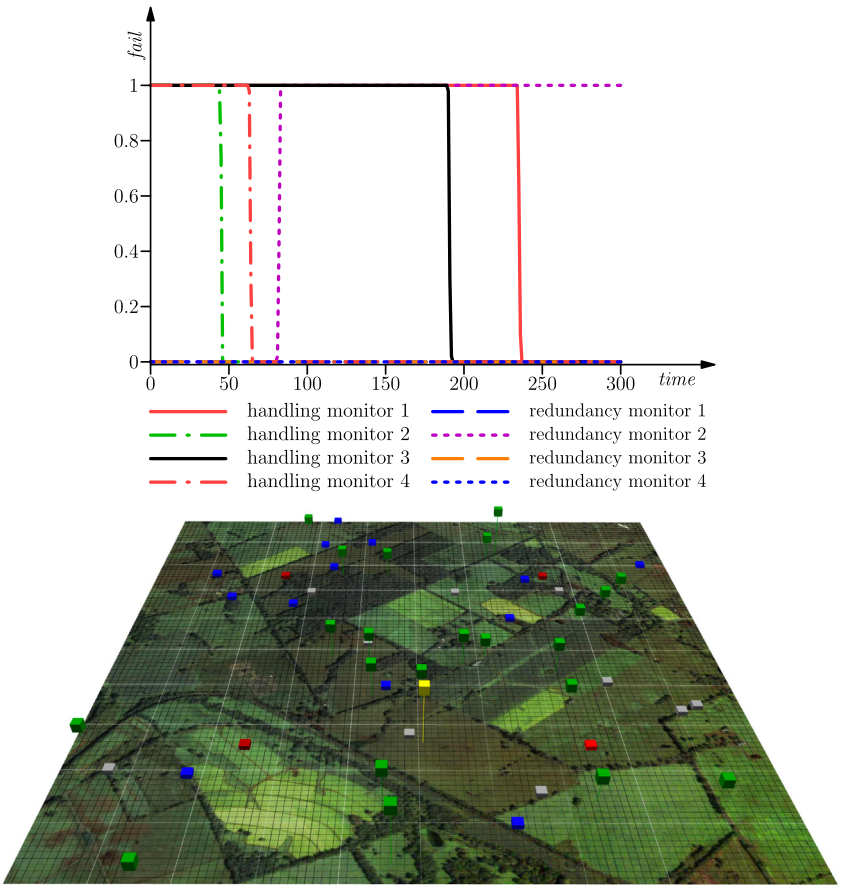

Fig. 10. Screenshot of a simulated drone area (bottom), monitors evolution in time (top). Brown cubes are communication towers (red if asking service), others are drones (green-moving, yellow-handling, blue-recharging, grey-idling). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

We evaluated this scenario in a simulated 500 m × 500 m area (see Fig. 11 (left) for a screenshot). We randomly choose 5 potential incident points (black cubes, that switch to red during an incident) and 100 people randomly walking through the area; the colour of the cubes representing people varies from green to blue (when they are *safe*) and from red to yellow (when they are *not safe*), depending on the distance from an incident. A person is safe when she is more than 50 m from the incident. The distance from the incident is not directly sensed by the people, but is computed through a distributed Field Calculus function that only assumes knowledge of the distance of a person from her neighbours. Units on people were able to communicate within a 50 m range, while incident points had an higher communication range (100 m with people, 200 m among them).

During the simulation, an incident happens in each of the 5 points, at a random time from 0 to 300 s, and lasts for a random duration from 0 to 100 s. Note that in this way incidents can, and in fact often do in practice, overlap with each other. Finally, an *alert* is perceived by people when they are within a 100 m radius from an incident. The reaction of a person being *unsafe* is to immediately start moving towards her neighbour that is further away from the incident, until she is at least 10 m outside the unsafe area, i.e., 60 m from the incident. Then, she goes back to random walk, which is an important factor for making it possible to violate local safety.

After the first incident at about 60 s, few people violate formula φ_1 of local safety. However, such a violation quickly spreads to most other people so that the indicator of *global* safety failure almost immediately reaches 100% (Fig. 11 (right), blue dashed line); this means that most or all of the people know that someone has behaved unsafely, i.e., entered the unsafe area after being in a safe place. The *local* safety failure indicator only increases when, during the five incidents, specific persons behave unsafely, and involves less than 10% of the people by the end of the simulation. Note that the introduction of the *all-paths* quantifier in this example creates the impression of a global observer, yet we have to recall that all preceding branches are derived from the event structure, and not every agent may be in contact with everyone else.

4.4. Smart home

Finally, we consider a smart home scenario borrowed from a preliminary work on using field calculus for runtime verification (Audrito et al., 2018b, Sect. 4.2), and hypothetically situated in a large building where multiple sensors are connected through a multi-hop network. Assume that we want to check that an electronic system (e.g. a room light) is active whenever some people have been present in the immediate vicinity in the close past.[8] Consider the predicates:

- s is true if the current device is responsible of an electronic system;
- a is true if s is true and the system is active;
- p is true if the current device is detecting the presence of people.

[8] In Audrito et al. (2018b), the property considered a single light active whenever people were present somewhere in the building.

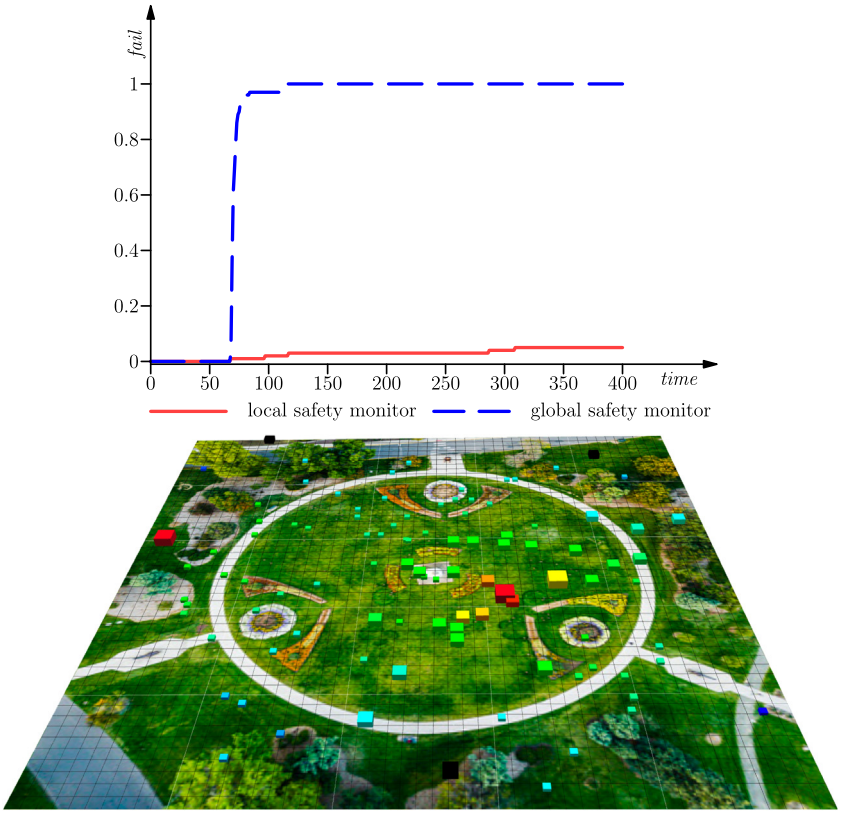

Fig. 11. Screenshot of a simulated incident area (bottom), monitors evolution in time (top). Incident points are black cubes (red during incidents), peoples' colour varies from green to blue (if safe) and from red to yellow (if unsafe) depending on distance. Cubes are larger where the monitor has failed. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The required property may be expressed as $\mathrm{AH}\,\phi_s$ where $\phi_s = s \rightarrow (p \leftrightarrow a)$, however, that would not accommodate for any (even small) response delay for the system. If we want to ensure that a is true iff p is true now and in the previous round (resp. with false), then we can change the property to $\mathrm{AH}\,\phi_w$ where:

$$\phi_w = s \rightarrow (p \wedge \mathrm{Y}\,p \rightarrow a) \wedge (\neg p \wedge \mathrm{Y}\,\neg p \rightarrow \neg a).$$

We evaluated this scenario in a simulated building floor comprising eight 6 m × 6 m rooms, each with a central light, around a 24 m × 3 m corridor with four lights (see Fig. 12 (left) for a screenshot). We populated the building with 12 people (pink cubes), randomly walking to a different room through the corridor after stops of 40 s in average (according to an exponential distribution), and assumed a short connection range of 3 m modelling a low-power Bluetooth system. Lights turned on (yellow) or off (grey) according to the presence of people within the connection range (predicate p), with a 5% chance of a delayed reaction during an on/off switch, and a lower 0.3% spurious reaction during rounds in which p was steady. Due to these error probabilities, the local strong (ϕ_s) and weak (ϕ_w) monitors became occasionally false on individual lights during the simulated time (small spikes in the graph). After the first error signalled by ϕ_s at about 1 s in the simulation, the global monitor $\mathrm{AH}\,\phi_s$ started becoming false; with the spreading of its value across the network being limited by the low connectivity of the network. A similar situation happened with ϕ_w and $\mathrm{AH}\,\phi_w$ at about 27 s. The screenshot represents an intermediate state, where some devices are aware of the failure (larger cubes) while other still are not yet.

5. Related work

5.1. Aggregate computing

The problem of finding suitable programming models for ensemble of devices has been the subject of intensive research—see e.g. the surveys (Beal et al., 2013; Viroli et al., 2019): works as TOTA (Mamei and Zambonelli, 2009) and Hood (Whitehouse et al., 2004) provide abstractions over the single device to facilitate construction of macro-level systems; GPL (Coore, 1999) and others are used to express spatial and geometric patterns; Regiment (Newton and Welsh, 2004) is an information system used to stream and summarise information over space–time regions; while MGS (Giavitto et al., 2004) and the fixpoint approach in Lluch-Lafuente et al. (2017) provide general purpose space–time computing models. Aggregate computing and the field calculus have then been developed as a generalisation of the above approaches, with the goal of defining a programming model with sufficient expressiveness to describe complex distributed processes by a functional-oriented compositional model, whose semantics is defined in terms of gossip-like computational processes.

Hence, *aggregate computing* (Beal et al., 2015) aims at supporting reusability and composability of collective adaptive behaviour as inherent properties. Following the inspiration of "fields" of physics (e.g., gravitational fields), this is achieved by the notion of *computational field* (simply called *field*) (Mamei and Zambonelli, 2009), defined as a global data structure mapping devices of the distributed systems to computational values. Computing with fields means deriving in a computable way an output field from

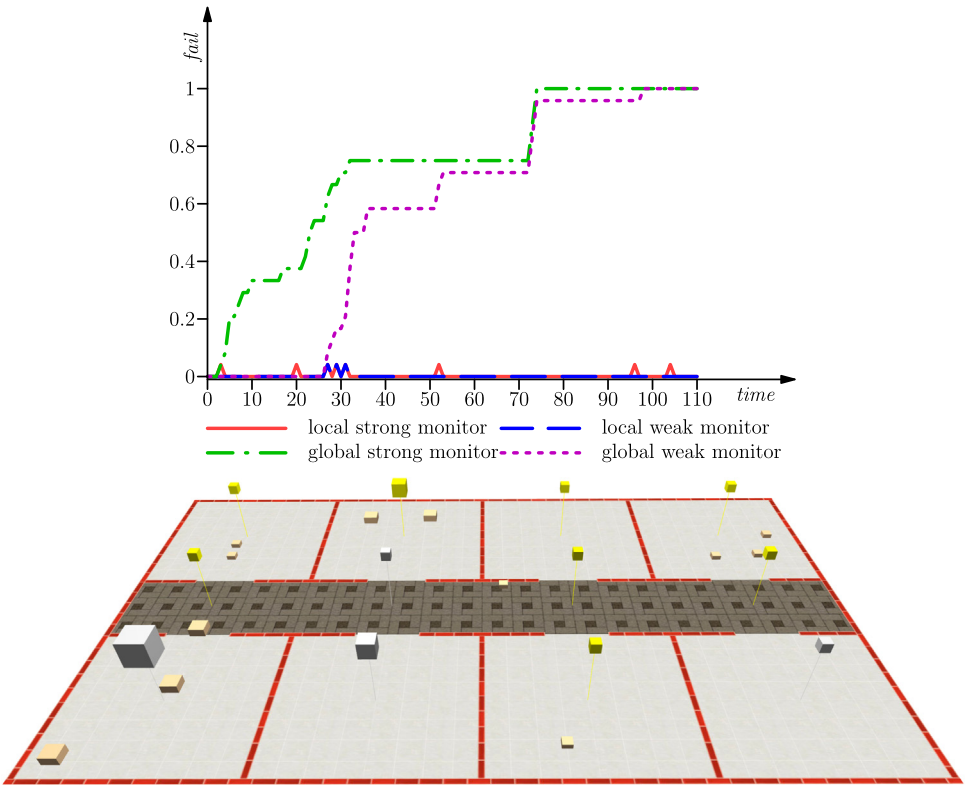

Fig. 12. Screenshot of a simulated building floor (bottom), monitors evolution in time (top). Lights are higher cubes (grey if off, yellow if on), lower cubes are people (larger cubes where the monitor has failed). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

a set of input fields. This can be done at a low-level, to defined programming language constructs or general-purpose building blocks or reusable behaviour, or at a high-level to design collective adaptive services or whole distributed applications—which ultimately work by getting input fields from sensors and process them to produce output fields to actuators.

The *field calculus* (Audrito et al., 2019) is a minimal functional language that identifies basic constructs to manipulate fields, and whose operational semantics can act as blueprint for developing toolchains to design and deploy systems of possibly myriad devices interacting via proximity-based broadcasts—see e.g. the FCPP, Protelis and ScaFi DSLs. The field calculus have been used to formally prove distributed systems properties of resiliency to environment changes (Audrito et al., 2018c; Nishiwaki, 2016; Viroli et al., 2018) and to device distribution (Beal et al., 2017), which makes it particular suitable for distributed runtime verification.

5.2. Distributed runtime verification

Runtime verification is a lightweight verification technique concerned with observing the execution of a system with respect to a specification (Leucker and Schallhart, 2009). Specifications are generally trace- or stream-based, with events that are mapped to atomic propositions in the underlying logic of the specification language. Popular specification languages include variations on the Linear Time Logic (LTL), and regular expressions, which can be effectively checked through finite automata constructions. Events may be generated through state changes or execution flow, such as method calls.

In *distributed* runtime verification, this concept is lifted to distributed systems, to meet applications in areas like: *(i)* observing distributed computations and expressiveness (specifications over the distributed systems), *(ii)* analysis decomposition (coupled composition of system- and monitoring components), *(iii)* exploiting parallelism (in the evaluation of monitors), *(iv)* fault tolerance and *(v)* efficiency gains (by optimising communication) (Francalanza et al., 2018).

Naturally, such lifting also affects the specification language. Bauer and Falcone (Bauer and Falcone, 2016) show a decentralised monitoring approach where disjoint atomic propositions in a global LTL property are monitored without a central observer in their respective components. Communication overhead is shown to be lower than the number of messages that would need to be sent to a central observer.

Sen et al. introduce PT-DTL (Sen et al., 2004) to specify distributed properties in a past time temporal logic. Subformulas in a specification are explicitly annotated with the node (or process) where the subformula should be evaluated. Communication of results of subcomputation is handled by message passing. A future time logic with explicit annotations (and a three-valued logic) based on the same idea has been presented in Scheffel and Schmitz (2014).

The above approaches assume a total communication topology, i.e., each node can send messages to everyone in the system, although causally unrelated messages may arrive in arbitrary order. Given the dynamic nature of IoT and edge scenarios, such a fixed distribution of verification sub-tasks is not possible in our approach, and the hierarchical structure does not further our aim of making each node an instance of the monitor.

As any other software engineering technique, also runtime verification in general has been applied in some form to IoT and edge computing. Tsigkanos et al. (2021) conduct a case-study in monitoring edge-based systems with temporal MFOTL specification in a similar explicit hierarchical setup. Unlike our approach, their network configuration is static and each tier

evaluates its own specification, and specifications explicitly refer to subformulas on other tiers. The latter distinction is only minor: our single field calculus specification would guard subformulas for the tier they apply to, and at the same time solve the problem of keeping signatures across formulas in sync.

Inçki and Ari (2018) specify expected behaviour in an IoT system through message sequence charts and then derive a monitor over traces that is centrally checked with a complex event processing tool. Unlike our solution, their approach relies on having network sniffers observing messages in each node and sending events to the central monitor.

We position ourselves in the following in the design space of distributed and decentralised runtime verification (see Francalanza et al., 2018): we assume that every agent, as a constituent of the whole system, executes independently and occasionally synchronises or communicates with other agents via the underlying communication platform. Going with the analogy of taking an agent as a *process* in Francalanza's terminology, we consider any two processes as *remote* to each other. 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 processes 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. Although the event structure may give the impression of a post-hoc view, the evaluation strategy that we have proposed is decidedly online: the field calculus expressions that evaluate branching expressions rely on communication with neighbouring agents (at that point in time).

Monitoring is performed by computation entities, identified by *monitors M*, that check properties of the system under analysis by analysing the traces. Similar to processes each monitor is hosted at a given location and may communicate with other monitors. As every agents is executing the same field calculus program, and hence the same monitor, our proposed approach brings a conceptual simplification. Namely, despite the distributed setting it resembles more the traditional setup of runtime verification: from the perspective of the programmer there is a single monitor and a single trace is evaluated—though this monitor is deployed and executed on a network of distributed devices and this trace contain events from different devices.

Moreover, being based on the field calculus, our approach is intrinsically resilient—cf. the discussion at the end of Section 5.1. Namely, it self-adapts to changes in the network topology. So, it automatically handles *failures*, which usually make distributed systems harder to manage: a non-responsive node (through a crash or through intermittently unavailable communication) only affects others through its lack in participation in evaluation of field expressions, which may affect the verdicts computed by its neighbours. We do not explicitly consider faulty sensors or message corruption, but leave this to integrity measures e.g. on the communication layer.

The absence of a global verdict, and instead having a per-node instantiation also puts us at odds with respect to another recent classification effort of distributed and decentralised runtime verification (Sánchez et al., 2019; El-Hokayem and Falcone, 2020). Such work considers challenges and approaches to monitor decomposition, placement and control (which does not apply here), and fault tolerance, for which we have made the case in the introduction that our approach presents a natural solution to. Many of the known issues in distributed runtime verification do not arise in our approach, due to the conceptual restriction of having to run the same monitor on every node.

6. Conclusion and future work

We provided a compositional translation of properties expressed in past-CTL logic into field calculus programs for monitoring them, proving that the translated monitor runs using local memory, message size and computation time that are all linear in the size of the formula (1 bit per temporal connective), matching the efficiency of previous result for past-LTL in a non-distributed setting (Havelund and Rosu, 2002). We then provided sample applications from different scenarios of edge computing (disaster management, swarm control, smart home), validating the monitor effectiveness by simulation under faults and changes, and arguing that past-CTL logic is reasonably expressive for properties of distributed systems.

In future work, we plan to extend this evaluation to scenarios based on real GPS traces, possibly expanding existing case studies in built environments (Audrito et al., 2021b), also considering the use of past-CTL properties in conjunction with runtime *reflection* (Leucker and Schallhart, 2009), where a property is annotated with a desired action (such as activation of an actuator) when it is activated, in order to manipulate the behaviour of agents for controlling the overall system evolution.

Furthermore, parallel studies have been investigating the usage of field calculus for monitoring SLCS spatial logic formulas (Audrito et al., 2021a). Despite aiming towards a similar purpose, this work shares very little with the present one: the two logics have no modal operators in common, and the models for interpreting them are incompatible. While past-CTL formulas are interpreted in event structures, SLCS formulas are interpreted in *graphs* (sets of nodes and vertices) without a temporal component, which are intended to represent the status of an evolving network in a particular instant. Even though time is abstracted away in the model of SLCS formulas, it still plays an implicit role, since the computation of the value of such formulas requires some time to converge and adjust. Expanding on a recent preliminary discussion on this topic (Audrito and Torta, 2021), in future work we plan to formalise the impact of time on SLCS formulas, giving an interpretation of them in event structures and thus allowing its merger with past-CTL, obtaining a full-fledged spatio-temporal logic.

Finally, we plan to investigate whether a distributed *continuation* operator could be devised to allow the monitoring of a future LTL-like logic in field calculus, in order to address settings where stronger prediction capabilities are needed and resources are not tightly constrained.

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.

References

Audrito, G., 2020. FCPP: an efficient and extensible field calculus framework. In: International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). IEEE, 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., Stolz, V., Viroli, M., 2021a. 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., Damiani, F., Giuda, G.M.D., Meschini, S., Pellegrini, L., Seghezzi, E., Tagliabue, L.C., Testa, L., Torta, G., 2021b. RM for users' safety and security in the built environment. In: Proceedings of the 5th ACM International Workshop on Verification and Monitoring at Runtime Execution (VORTEX). ACM, pp. 13–16. http://dx.doi.org/10.1145/3464974.3468445.

Audrito, G., Damiani, F., Stolz, V., Viroli, M., 2018b. On distributed runtime verification by aggregate computing. In: 2nd Workshop on Verification of Objects at RunTime Execution (VORTEX). In: EPTCS, vol. 302, pp. 47–61. http://dx.doi.org/10.4204/EPTCS.302.4.

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

Audrito, G., Torta, G., 2021. Towards aggregate monitoring of spatio-temporal properties. In: Proceedings of the 5th ACM International Workshop on Verification and Monitoring at Runtime Execution (VORTEX). ACM, pp. 26–29. http://dx.doi.org/10.1145/3464974.3468448.

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.

Bauer, A., Falcone, Y., 2016. Decentralised LTL monitoring. Formal Methods Syst. Des. 48 (1–2), 46–93. http://dx.doi.org/10.1007/s10703-016-0253-8.

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), 22–30. 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. Autonomous Adaptive Syst. (ISSN: 1556-4665) 12 (3), 12:1–12:29. http://dx.doi.org/10.1145/3105758.

Casadei, R., Viroli, M., Audrito, G., Damiani, F., 2020. Fscafi : A core calculus for collective adaptive systems programming. In: Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles. In: Lecture Notes in Computer Science, vol. 12477, Springer, pp. 344–360. http://dx.doi.org/10.1007/978-3-030-61470-6_21.

Coore, D., 1999. Botanical Computing: A Developmental Approach to Generating Inter connect Topologies on an Amorphous Computer (Ph.D. thesis). MIT, Cambridge, MA, USA.

El-Hokayem, A., Falcone, Y., 2020. On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Trans. Softw. Eng. Methodol. 29 (1), http://dx.doi.org/10.1145/3355181.

Francalanza, A., Pérez, J.A., Sánchez, C., 2018. Runtime verification for decentralised and distributed systems. In: Lectures on Runtime Verification - Introductory and Advanced Topics. In: Lecture Notes in Computer Science, vol. 10457, Springer, pp. 176–210. http://dx.doi.org/10.1007/978-3-319-75632-5_6.

Giavitto, J., Michel, O., Cohen, J., Spicher, A., 2004. Computations in space and space in computations. In: Unconventional Programming Paradigms. In: Lecture Notes in Computer Science, vol. 3566, Springer, pp. 137–152. http://dx.doi.org/10.1007/11527800_11.

Gigante, N., Montanari, A., Reynolds, M., 2017. A one-pass tree-shaped tableau for LTL+Past. In: Logic For Programming, Artificial Intelligence and Reasoning (LPAR). In: EPiC Series in Computing, vol. 46, EasyChair, pp. 456–473, URL http://www.easychair.org/publications/paper/340363.

Havelund, K., Rosu, G., 2002. Synthesizing monitors for safety properties. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). In: Lecture Notes in Computer Science, vol. 2280, Springer, pp. 342–356. http://dx.doi.org/10.1007/3-540-46002-0_24.

Inçki, K., Ari, I., 2018. A novel runtime verification solution for IoT systems. IEEE Access 6, 13501–13512. http://dx.doi.org/10.1109/ACCESS.2018.2813887.

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.

Leucker, M., Schallhart, C., 2009. A brief account of runtime verification. J. Log. Algebr. Program. 78 (5), 293–303. http://dx.doi.org/10.1016/j.jlap.2008.08.004.

Levine, R.V., Norenzayan, A., 1999. The pace of life in 31 countries. J. Cross-Cultural Psychol. 30 (2), 178–205, Ch. 16.

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

Mamei, M., Zambonelli, F., 2009. Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. Softw. Eng. Methodol. 18 (4), 1–56. http://dx.doi.org/10.1145/1538942.1538945.

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.

Nishiwaki, Y., 2016. F-Calculus: A universal programming language of self-stabilizing computational fields. In: 1st Intl. Workshops on Foundations and Applications of Self* Systems (FAS*W). IEEE, pp. 198–203. http://dx.doi.org/10.1109/FAS-W.2016.51.

Pianini, D., Viroli, M., Beal, J., 2015. Protelis: practical aggregate programming. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. ACM, pp. 1846–1853. http://dx.doi.org/10.1145/2695664.2695913.

Sánchez, C., Schneider, G., Ahrendt, W., Bartocci, E., Bianculli, D., Colombo, C., Falcone, Y., Francalanza, A., Krstic, S., Lourenço, J.M., Nickovic, D., Pace, G.J., Rufino, J., Signoles, J., Traytel, D., Weiss, A., 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). Formal Methods Syst. Des. 54 (3), 279–335. http://dx.doi.org/10.1007/s10703-019-00337-w.

Scheffel, T., Schmitz, M., 2014. Three-valued asynchronous distributed runtime verification. In: Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE).

Sen, K., Vardhan, A., Agha, G., Rosu, G., 2004. Efficient decentralized monitoring of safety in distributed systems. In: 26th Intl. Conf. on Software Engineering. pp. 418–427. http://dx.doi.org/10.1109/ICSE.2004.1317464.

Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L., 2016. Edge computing: Vision and challenges. IEEE Internet Things J. 3 (5), 637–646.

Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z., Stankovski, V., 2018. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. J. Syst. Softw. 136.

Tsigkanos, C., Bersani, M.M., Frangoudis, P.A., Dustdar, S., 2021. Edge-based runtime verification for the internet of things. IEEE Trans. Services Comput. 1.

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. 109, http://dx.doi.org/10.1016/j.jlamp.2019.100486,

Whitehouse, K., Sharp, C., Culler, D.E., Brewer, E.A., 2004. Hood: A neighborhood abstraction for sensor networks. In: Banavar, G.S., Zwaenepoel, W., Terry, D., Want, R. (Eds.), 2nd International Conference on Mobile Systems, Applications, and Services. ACM / USENIX, http://dx.doi.org/10.1145/990064.990079.