Adaptive distributed monitors of spatial properties for cyber–physical systems

Adaptive distributed monitors of spatial properties for cyber-physical systems

Giorgio Audrito[a], Roberto Casadei[b], Ferruccio Damiani[a], Volker Stolz[c], Mirko Viroli[b]

[a] *Dipartimento di Informatica, University of Turin, Turin, Italy*
[b] *DISI, Alma Mater Studiorum – Università di Bologna, Cesena, Italy*
[c] *Department of Computing, Mathematics and Physics, Western Norway University of Applied Sciences, Bergen, Norway*

Abstract

Cyber-physical systems increasingly feature highly-distributed and mobile deployments of devices spread over large physical environments: in these contexts, it is generally very difficult to engineer trustworthy critical services, mostly because formal methods generally hardly scale with the number of involved devices, especially when faults, continuous changes, and dynamic topologies are the norm. To start addressing this problem, in this paper we devise a formally correct and self-adaptive implementation of distributed monitors for spatial properties. We start from the Spatial Logic of Closure Spaces, and provide a compositional translation that takes a formula and yields a distributed program that provides runtime verification of its validity. Such programs are expressed in terms of the field calculus, a recently emerged computational model that focusses on global-level outcomes instead of single-device behaviour, and expresses distributed computations by pure functions and the functional composition mechanism. By reusing previous results and tools of the field calculus, we prove correctness of the translation, self-stabilisation of the derived monitors, and empirically evaluate adaptivity of such monitors in a realistic smart city scenario of safe crowd monitoring and control.

Keywords: Spatial Logics, Runtime verification, Self-adaptive systems, Field Calculus

1. Introduction

Cyber-physical systems (CPSs) are typically constructed by deploying a variety of computational devices of various sorts (sensors, actuators, computers)

⋆Corresponding author.
Email addresses: giorgio.audrito@unito.it (Giorgio Audrito⋆), roby.casadei@unibo.it (Roberto Casadei), ferruccio.damiani@unito.it (Ferruccio Damiani), volker.stolz@hvl.no (Volker Stolz), mirko.viroli@unibo.it (Mirko Viroli)

into the physical environment, e.g., in scenarios like smart cities, intelligent buildings and factories, transportation, and wide-area monitoring and control. Such systems increasingly feature large-scale, intrinsic distribution of computation, dynamism, mobility, and unpredictability due to faults, adversarial behaviour, and unknown patterns of human behaviour and data production. As such, engineering trustworthy computational services over CPSs is particularly challenging, especially when there is need of facing critical functional and non-functional requirements. In principle, one would seek for formal methods and tools by which a system design can be verified against the validity of certain properties, such that these properties can be transferred to system execution: unfortunately, the complexity of CPSs deployment scenarios typically makes the problem intractable (Bennaceur et al., 2019).

As a contribution towards facing this issue in a significant class of cases, in this paper we focus on the problem of decentralised distributed runtime verification of spatial properties. 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, so as to precisely state the properties to check as well as providing formal guarantees about the results of monitoring (Bauer et al., 2011; Leucker and Schallhart, 2009): distributed runtime verification is runtime verification in connection with distributed systems, hence it comprises both monitoring of distributed systems and using distributed systems for monitoring in an asynchronous setting. Approaches to distributed runtime verification typically rely on simplifying assumptions such as absence of failures and mobility (Francalanza et al., 2018). Here, we specifically aim at open systems of agents, where the number of participants, their communication topology, and the performance of (broadcast) messages is unreliable. According to the above survey, there is no comparative work in this regard.

In a further departure from other approaches to runtime verification, we are specifically not interested in a global verdict (and hence a global monitor), but rather in each agent's view, which in a slight departure from the terminology we call the decentralised setting. In our application area we specifically want to avoid a centralised monitor (observer), as nodes e.g. in a wireless sensor network can only communicate with neighbours, and we see distribution also as a way to tolerate failures such as network partitioning, in addition to a pragmatic motivation.

We address this problem of an open system by careful selection of a computational model for distributed systems that provides inherent support to large-scale and open scenarios, scalability with the complexity of the distributed algorithms to implement, and compositionality with respect to spatial logical connectives. Namely, we adopt the aggregate computing paradigm (Beal et al., 2015; Viroli et al., 2019), and especially its incarnation into the field calculus language (Audrito et al., 2019), where agents are programmed in an abstract computational environment and make use also of spatial and temporal data constructs in their region through proximity-based interactions. Programs expressed in field calculus focus on global-level outcomes of a computation in-

stead of single-device behaviour, they express highly-distributed computations by pure functions, and finally rely on functional composition as key mechanism to combine libraries of reusable and correct building blocks into higher-level applications services Viroli et al. (2018). At the modelling level, the field calculus expresses computations as transformation 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. 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 minutes is a function from the temperature field to a field of Booleans.

We take as reference the Spatial Logic of Closure Spaces (SLCS) (Ciancia et al., 2014), a modal logic proposed to describe and verify topological properties over spatially-situated systems, and formally grounding the concepts of proximity, propagation and surroundedness. By the field calculus, we are able to define a translation of formulas of the Spatial Logic of Closure Spaces (SLCS) into distributed systems that act as monitors for such formulas, namely, making all nodes of the system collaborate by local interaction to establish the validity of the formula at each point of space. By reusing previous results and tools of the field calculus, then, we prove correctness of the translation, and self-stabilisation of the derived monitors, hence their robustness to transient changes. Finally, we empirically evaluate self-adaptivity of the generated monitors in a realistic smart city scenario of safe crowd monitoring. In such a case study, we consider a target SLCS property and compare a corresponding decentralised field calculus monitor (straightforwardly obtained by applying the formula mappings) with an ideal, oracle monitor having direct complete knowledge of the entire distributed system. We show that the former monitor, despite the adversarial conditions of nearly-continuous topology change, is able to approximate the ideal monitor with reasonable precision. This demonstrates the practical viability of the approach (though, of course, its suitability ultimately depends on the relative reactivity and precision requirements of the application at hand) as well as its significance especially in scenarios where centralised services are not practicable (e.g., because of missing infrastructure) or temporarily unavailable (c.f. graceful degradation).

The remainder of this paper is organised as follows: Section 2 provides the necessary background; Section 3 illustrates how the field calculus can be used to implement distributed monitors of spatial logic properties; Section 4 presents the case study; Section 5 discusses some related work; and Section 6 concludes.

2. Background

In this section we provide the necessary background to introduce the key contribution of the paper in next section. In particular, Section 2.1 describes distributed runtime verification and how our approach relates to it, Section 2.2 introduces aggregate computing and the field calculus, and finally Section 2.3 discusses spatial logics and introduces the SLCS logics.

2.1. 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. Events may be generated through state changes or execution flow, such as method calls.

Most specifically, this paper focusses on the sub-case of so-called distributed runtime verification (Francalanza et al., 2018), aiming at defining logics to express properties of space and time, and corresponding monitors for such properties—which may or may not be distributed. In distributed runtime verification, agents (representing the system to verify at each device) are generally considered *remote* to each other: as constituents of the whole system, they are assumed to execute independently and occasionally synchronise or communicate with each other via the underlying communication platform. A *local trace* of *events* corresponds to a sequence of sets of values for observables, as defined through the sensors of an agent, or derived values from those. Since agents may appear or disappear over time from the overall system, traces from different 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. Accordingly, logic formulas cannot state properties over single traces, but one should naturally adopt an "event structure" viewpoint (Winskel, 1982), where a partial order relation between events is introduced to model causality (communication across agents, or agent internal computation steps) (Audrito et al., 2019).

Monitoring is performed by computation entities that check properties of the system under analysis by analysing traces representing partial system evolutions. Similar to the agents carrying on system execution, each monitor is hosted at a given location and may communicate with other monitors, though in general there is no strict correspondence between locations of agents and monitors. Additionally, *failures*, such as lost or corrupt messages, are typically ignored, for they would make the overall distributed monitor unable to carry on the verification process in a meaningful way.

The distributed monitoring approach we shall introduce in this paper is in a sense a more natural transition from traditional runtime verification. It is based on the idea of locating monitors on each device and make them execute the same local program, which amounts to evaluate single traces that include all local events as well as events from neighbouring nodes (perceived by communication). Given that the formula of a spatial logic may have a validity result that varies at each point of space, the local monitor at a device will naturally give the validity result of its location: each local result, however, has been derived through the collaboration of all local monitors through broadcast communication, hence, in a distributed way. Accordingly, unexpected changes in the system configuration (node/device faults, changes in topology, permanent loss of communications),

affect the executing system in the same way they affect the distributed monitor, which will then behave accordingly and coherently. Namely, any change will be considered as a new network configuration, in which the distributed monitor will continue its verification process, taking into account the changed set of neighbours in subsequent communication rounds.

2.2. Aggregate computing

Aggregate computing (Beal et al., 2015; Viroli et al., 2019) aims at supporting reusability and composability of collective adaptive behaviour. 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 system to computational values. Computing with fields means deriving in a computable way an output field from a set of input fields. Field computations can be understood both *locally*, in terms of interactions with neighbours, or *globally* in terms of composition of functions on fields.

2.2.1. Computational model

In aggregate computing, the global evolution of a computing network is carried out by periodically and asynchronously executing on every device a same program `P` according to a cyclic schedule. Thus, every device δ in the network independently performs a sequence $\epsilon_1, \epsilon_2, \ldots$ of *firings*, each of them consisting of the following actions:

1. the device perceives contextual information formed by data provided by sensors, local information stored in the previous firing, and messages recently collected from neighbours[1] (discarding older messages after a certain timeout), the latter in the form of a so-called *neighbouring value* ϕ—essentially a map $\phi = \delta_1 \mapsto \mathtt{v}_1, \ldots, \delta_n \mapsto \mathtt{v}_n$ ($n \geq 1$) from neighbour devices' identifiers $\delta_1, \ldots, \delta_n$ to corresponding values $\mathtt{v}_1, \ldots, \mathtt{v}_n$.

2. then, the device evaluates the program `P`, considering as input the contextual information gathered as described above;

3. the result of the local computation is a data structure that is stored locally, broadcast to neighbours, and possibly fed to actuators producing output values;

4. finally, the device goes to sleep waiting for its next firing, while gathering messages from neighbours.

Firings and their mutual relationships are modelled formally through the established notion of *event structures* (Lamport, 1978) and its *augmented* variant with device identifiers (Audrito et al., 2018a, 2019). This representation focuses

[1] 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.

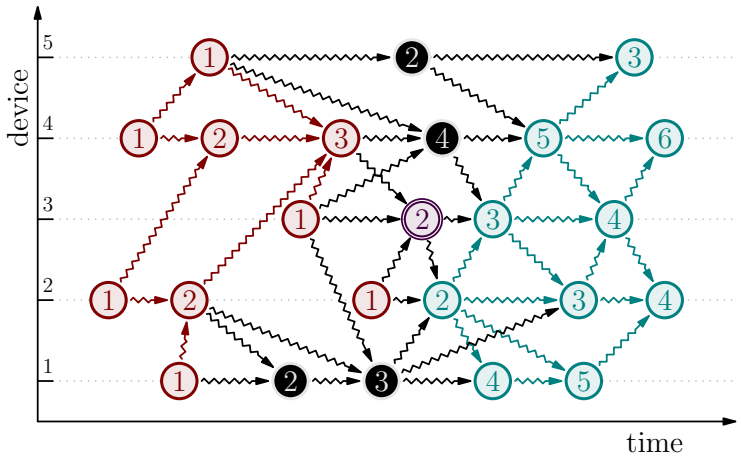


Figure 1: Example of an event structure (which is also a LUIC augmented event structure, c.f. Def.2), comprising events (circles), neighbour relations (arrows), devices (ordinate axis). Colours indicate causal structure with respect to the doubly-circled event (magenta), splitting events into causal past (red), causal future (cyan) and concurrent (non-ordered, in black). The numbers written within events represent a sample space-time value (c.f. Def. 3) associated with that event structure. Note that the doubly-circled event has three neighbouring events: event 1 at the same device (its previous firing), event 3 at device 4, and event 1 at device 2. Figure taken from (Audrito et al., 2018a).

on the communication between devices, which is the main aspect distinguishing a distributed system from a sequential one: the relation of this representation with the physical neighbour-relationship of devices at each point in time will be discussed later (c.f. Section 2.2.2).

Definition 1 (Event Structure)**.** An *event structure* $\langle E, \rightsquigarrow, < \rangle$ is a finite or countably infinite[2] set of *events* E together with a neighbouring relation $\rightsquigarrow \subseteq E \times E$ and a causality relation $< \subseteq E \times E$, such that the transitive closure of $\rightsquigarrow$ forms the irreflexive partial order $<$, and the set $X_\epsilon = \{\epsilon' \in E \mid \epsilon' < \epsilon\} \cup \{\epsilon' \in E \mid \epsilon \rightsquigarrow \epsilon'\}$[3] is finite for all ϵ (i.e., $<$ and $\rightsquigarrow$ are locally finite).

Thus, we say that ϵ' is a neighbour of ϵ iff $\epsilon' \rightsquigarrow \epsilon$, and that $\mathcal{N}(\epsilon) = \{\epsilon' \in E \mid \epsilon' \rightsquigarrow \epsilon\}$ is the set of neighbours of ϵ.

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

[2] Although infinite event structures are not a representation of a realistic system execution, they are a useful theoretical tool to study eventual behaviour of networks.

[3] In the remainder of this paper, we use the infix notation for binary relations, so that e.g. $\epsilon \rightsquigarrow \epsilon'$ stands for $(\epsilon, \epsilon') \in \rightsquigarrow$.

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

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

Definition 2 (LUIC Augmented Event Structure)**.** An *augmented event structure* is a tuple $\mathbf{E} = \langle E, \rightsquigarrow, <, d\rangle$ 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:

- $\text{next} : E \twoheadrightarrow E$ as the partial function[4] 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 not $\epsilon' \rightsquigarrow \epsilon$.

We say that $\mathbf{E}$ 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);
- **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.

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

(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 modeled as happening instantaneously. In this scenario, the explicit causal dependencies imply additional time dependencies $\epsilon \dashrightarrow \epsilon'$: if ϵ' was able to reach $\mathrm{next}(\epsilon)$ but not ϵ, the firing of ϵ' must have happened *after* ϵ (additional details on this point may be found in the proof of Theorem 1 in Appendix B.1).

Remark 1 (On Augmented Event Structures). Augmented event structures were first implicitly used in Audrito et al. (2019) for defining the denotational semantics (with the *linearity* and *uniqueness* constraints only), then formalised in Audrito et al. (2018a) (without any explicit constraint embedded in the definition). In this paper, we gathered all necessary constraints to capture exactly which augmented event structures correspond to physically plausible executions of an aggregate system (see Theorem 1): this includes both the *linearity* and *uniqueness* from Audrito et al. (2019), together with the new *impersistence* and *computation immediacy* constraints.

Notice that the event structure in Figure 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. Notice that 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).

Through the repetitive execution of firings (modelled by events), across space (where devices are located) and time (when devices fire), a global behaviour emerges. This global behaviour is defined in terms of global data structures called *space-time values* (also depicted in Figure 1) mapping events to values for each event in an event structure.

Definition 3 (Space-Time Value)**.** Let $\mathbf{V}$ be any domain of computational values and $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be an augmented event structure. A space-time value $\Phi = \langle \mathbf{E}, f \rangle$ is a pair comprising the event structure and a function $f : E \to \mathbf{V}$ that maps the events $\epsilon \in E$ to values $\mathtt{v} \in \mathbf{V}$. With abuse of notation, in the remainder of this paper we use $\Phi(\epsilon)$ to denote $f(\epsilon)$ where $\Phi = \langle E, f \rangle$.

A space-time value Φ represents a quantity that is distributed across space end evolving through time, so that its value $\Phi(\epsilon)$ may be different on different events ϵ. For example, Φ may associate events to the corresponding measurements of a sensor $\Phi(\epsilon)$ available in them, or to the current local result $\Phi(\epsilon)$ of a distributed computation. 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 4 (Space-Time Function)**.** Let $\mathbf{V}(\mathbf{E}) = \{\langle \mathbf{E}, f\rangle \mid \ f : E \to \mathbf{V}\}$ be the set of all possible space-time values in a augmented event structure $\mathbf{E}$. Then, an *n-ary space-time function in* $\mathbf{E}$ is a partial map $\mathtt{f} : \mathbf{V}(\mathbf{E})^n \twoheadrightarrow \mathbf{V}(\mathbf{E})$.

Notice that the definition of a space-time function `f` requires every input and output space-time value to exist in the same augmented event structure $\mathbf{E}$. 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 (c.f. Section 2.2.3 for such a programming language), 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 an operational semantics, see Appendix A).

2.2.2. Stabilisation and spatial model

Even though the global interpretation of a program has to be given in spatio-temporal terms in general, for a relevant class of programs a space-only representation is also possible. In this representation, *event structures*, *space-time values* and *space-time functions* are replaced by *network graphs*, *computational fields* and *field functions*.

Definition 5 (Network Graph)**.** A *network graph* $\mathbf{G} = \langle D, \rightarrowtail\rangle$ is a finite set D of *devices* δ together with a reflexive neighbouring relation $\rightarrowtail \subseteq D \times D$, i.e., such that $\delta \rightarrowtail \delta$ for each $\delta \in D$. Thus, we say that δ' is a neighbour of δ iff $\delta' \rightarrowtail \delta$, and that $\mathcal{N}(\delta) = \{\delta' \in D \mid \ \delta' \rightarrowtail \delta\}$ is the set of neighbours of δ.

Intuitively, an instance element $\delta_1 \rightarrowtail \delta_2$ that belongs to the neighbouring relation $\rightarrowtail$ on devices (in a certain specific instant of time) represents the possibility for a device δ_1 to successfully send a message to another device δ_2, thus creating corresponding instance elements $\epsilon_1 \rightsquigarrow \epsilon_2$ of the neighbouring relation $\rightsquigarrow$ on events for events ϵ_i on devices δ_i. Notice that $\rightarrowtail$ does not necessarily have to be symmetric, since e.g. an high-power device may be able to send messages to a distant device with not enough power to reply. The formal relationship between relations $\rightarrowtail$ and $\rightsquigarrow$ will be captured by Definition 8 below.

Definition 6 (Computational Field)**.** Let $\mathbf{V}$ be any domain of computational values and $\mathbf{G} = \langle D, \rightarrowtail\rangle$ be a network graph. A computational field $\Psi = \langle \mathbf{G}, g\rangle$

is a pair comprising the network graph and a function $g : D \to \mathbf{V}$ mapping devices $\delta \in D$ to values $\mathtt{v} \in \mathbf{V}$.

Definition 7 (Field Function)**.** Let $\mathbf{V}(\mathbf{G}) = \{\langle \mathbf{G}, g\rangle \mid \ g : D \to \mathbf{V}\}$ be the set of all possible computational fields in a network graph G. Then, an *n-ary field function in G* is a partial map $\mathtt{g} : \mathbf{V}(\mathbf{G})^n \twoheadrightarrow \mathbf{V}(\mathbf{G})$.

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

Definition 8 (Stabilising Augmented Event Structure and Limit)**.** Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d\rangle$ be a countably infinite augmented event structure. We say that $\mathbf{E}$ is *stabilising* to its limit $\mathbf{G} = \langle D, \rightarrowtail\rangle = \lim \mathbf{E}$ iff $D = \{\delta \mid \ \{\epsilon \in E.\ d(\epsilon) = \delta\} \text{ is infinite}\}$ is the set of devices appearing infinitely often in $\mathbf{E}$, and for all except finitely many $\epsilon \in E$, the devices of neighbours are the neighbours of the device of ϵ:

$$\{d(\epsilon') \mid \ \epsilon' \rightsquigarrow \epsilon\} = \{\delta' \mid \ \delta' \rightarrowtail d(\epsilon)\}$$

Although the notion of *limit of an event structure* provided by Definition 8, it is not identical to the notion of *limit* in analysis; they are intimately connected, justifying the usage of the same name. In concrete deployments, the augmented event structure representing a distributed computation performed over time and across space arises from a network graph (evolving over time) which represents the possible connections across devices in every instant of time. Then, the *limit* of the event structure is (intuitively) the network graph that is obtained for the time that goes to infinity. A similar notion of limit (and stabilisation) can also be applied to space-time values as shown in the following.

Definition 9 (Stabilising Space-Time Value and Limit)**.** Let $\Phi = \langle \mathbf{E}, f\rangle$ be a space-time value on a stabilising augmented event structure $\mathbf{E} = \langle E, \rightsquigarrow, <, d\rangle$ with limit $\mathbf{G} = \lim \mathbf{E}$. We say that Φ is stabilising to its limit $\Psi = \langle \mathbf{G}, g\rangle = \lim \Phi$ iff for all except finitely many $\epsilon \in E$, $f(\epsilon) = g(d(\epsilon))$.

Notice that $\mathbf{G}$ is not a parameter of the definition above, by being uniquely determined by $\mathbf{E}$.

Definition 10 (Self-Stabilising Space-Time Function and Limit)**.** Let $\mathtt{f} : \mathbf{V}(\mathbf{E})^n \twoheadrightarrow \mathbf{V}(\mathbf{E})$ be an n-ary space-time function in a stabilising $\mathbf{E}$ with limit $\mathbf{G}$. We say that $\mathtt{f}$ is *self-stabilising* with limit $\mathtt{g} : \mathbf{V}(\mathbf{G})^n \twoheadrightarrow \mathbf{V}(\mathbf{G})$ iff for any $\langle \Phi_1, \ldots, \Phi_n\rangle$ with limit $\langle \Psi_1, \ldots, \Psi_n\rangle$, $\mathtt{f}(\Phi_1, \ldots, \Phi_n) = \Phi$ with limit $\Psi = \mathtt{g}(\Psi_1, \ldots, \Psi_n) = \lim \Phi$.

Many of the most commonly used routines in aggregate computing are indeed self-stabilising, and in fact belong to a class of self-stabilising functions identified in Viroli et al. (2018). In the remainder of this paper, we shall relate aggregate functions with spatial logical formulas, expressing their relationship in terms of their self-stabilising limit (see Theorem 3).

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

Figure 2: Syntax of the field calculus language.

2.2.3. The field calculus

The *field calculus* (Audrito et al., 2019) is a minimal functional language that identifies basic constructs to manipulate aggregate fields, and whose operational semantics can act as blueprint for developing toolchains to design and deploy systems of possibly myriad devices interacting via local (e.g., proximity-based) broadcasts. The field calculus provides the necessary mechanisms to express and 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. Each field calculus function comes with both a global interpretation (through a denotational semantics in terms of space-time functions and/or field functions (Audrito et al., 2019; Viroli et al., 2019)), and a local interpretation (through an operational semantics defining the operations performed in a firing, see Appendix A) which provides a practical and correct implementation of the global interpretation. These interpretations are so that *compositionality* holds, that is, function composition in the language translate to space-time function composition.

The syntax of field calculus is presented in Figure 2, following the presentation by Audrito et al. (2020), simplified to fit the needs of this paper. The overbar notation $\overline{\texttt{e}}$ is a shorthand for sequences of elements, and multiple overbars are intended to be expanded together, e.g., $\overline{\texttt{e}}$ stands for $\texttt{e}_1, \ldots, \texttt{e}_n$ and $\overline{\delta} \mapsto \overline{\ell}$ for $\delta_1 \mapsto \ell_1, \ldots, \delta_n \mapsto \ell_n$. Operator `share` and `nbr` are the main peculiar constructs of the field calculus, the former responsible for both interaction and field dynamics, the latter for observing neighbouring values; while `def` and `if` correspond to the standard function definition and the branching expression constructs, properly adapted to fit computational fields. We remark that the field calculus comes with a type system, which performs standard checks on function calls, ensures that the arguments of `share` and `nbr` have the proper type, and guards of branches have Boolean type—a detailed presentation of the field calculus type system can be found in Audrito et al. (2019). In this paper, we use `bool` (Boolean values) and `num` (numbers with infinity) as primitive types T, together with types $\texttt{field}[T]$ for neighbouring fields built from values of type T, and types $(\overline{T}) \to T$ for functions.

A program `P` consists of a list of function definitions $\overline{\texttt{F}}$, 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 firing (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 or a list).
  - A *neighbouring value* ϕ that associates neighbour devices δ to local values ℓ, e.g., it could be the neighbouring value of distances of neighbours—note that neighbouring 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. The built-in functions and data constructors used in this paper are listed in Figure 3 together with their types and formal interpretations—all of these operators are natively available in existing implementations of the field calculus, including Protelis (used in the case study presented in Section 4).
- A branching expression `if`($\mathtt{e}_1$){$\mathtt{e}_2$}`else`{$\mathtt{e}_3$}, used to split 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 neighbouring expression `nbr`{`e`}, where `e` evaluates to a local value. It evaluates to a neighbouring value mapping neighbours to their latest available result of evaluating `e`. Each device δ:
  1. shares its value of `e` with its neighbours, and
  2. evaluates the expression into a neighbouring value ϕ mapping each neighbour δ' of δ to the latest value that δ' has shared for `e`.

  Note that within an `if` branch, `nbr` is restricted to work on device events within the subspace of the branch. I.e., the evaluation by a device δ of an `nbr`-expression within a branch of some `if`($\mathtt{e}_1$){...}`else`{...} expression, is affected only by the neighbours of δ that, during their last computation cycle, evaluated the same value for the guard $\mathtt{e}_1$ (domain restriction, Audrito et al., 2016).

Constructors:		
`true`, `false`	$= () \to \texttt{bool}$	$\top,\ \bot$
`0`, `1`, `infinity`	$= () \to \texttt{num}$	$0,\ 1,\ \infty$
Built-ins:		
`!`	$= (\texttt{bool}) \to \texttt{bool}$	$\neg$
`||`, `&&`	$= (\texttt{bool}, \texttt{bool}) \to \texttt{bool}$	$\vee,\ \wedge$
`<=`, `==`	$= (T, T) \to T$ for $T \in \texttt{bool}, \texttt{num}$	$\leq,\ =$
`+`, `-`	$= (\texttt{num}, \texttt{num}) \to \texttt{num}$	$+,\ -$
`mux`	$= (\texttt{bool}, \texttt{num}, \texttt{num}) \to \texttt{num}$	$(b, x, y) \mapsto x$ if b else y
`minHood`	$= (\texttt{field}[\texttt{num}]) \to \texttt{num}$	$\phi \mapsto \min \{\phi(\delta') \mid \delta' \in \mathbf{dom}(\phi) \setminus \{\delta\}\}$
`anyHoodPlusSelf`	$= (\texttt{field}[\texttt{bool}]) \to \texttt{bool}$	$\phi \mapsto \bigvee \{\phi(\delta') \mid \delta' \in \mathbf{dom}(\phi)\}$
`allHoodPlusSelf`	$= (\texttt{field}[\texttt{bool}]) \to \texttt{bool}$	$\phi \mapsto \bigwedge \{\phi(\delta') \mid \delta' \in \mathbf{dom}(\phi)\}$

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

- A share expression $\texttt{e} = \texttt{share}(\texttt{e}_1)\{(\texttt{x})\texttt{=>}\texttt{e}_2\}$, where $\texttt{e}_1$ and $\texttt{e}_2$ evaluate to a local values. It incorporates message passing and local state evolution. The result of such expression is obtained by:
  - Gathering the results obtained by neighbours for the whole expression `e` in their last firings into a neighbouring field value ϕ.
  - Such ϕ may also contain a value for the current device (if it is not the first firing it executes `e`). If not, the result of evaluating $\texttt{e}_1$ is used as value for the current device and incorporated into ϕ.
  - Expression $\texttt{e}_2$ is evaluated by substituting ϕ to `x`, obtaining the overall value `v` for `e`.
  - Value `v` is broadcast to neighbours, allowing them to use it in constructing their following neighbours' observation ϕ.

  Note that within an `if` branch, `share` (like `nbr` as described above) is restricted to work on device events within the subspace of the branch. This construct can be used both for structuring device interaction and for evolving a local state, by using the built-in operator $\texttt{locHood}(\phi)$ which extracts the value $\phi(\delta)$ relative to the current device from a neighbouring field value ϕ.

Notice that δ (the current device where the computation takes place) is *excluded* from `minHood` but *included* in `anyHoodPlusSelf` and `allHoodPlusSelf`; and that $\texttt{mux}(\texttt{e}, \texttt{e}_\top, \texttt{e}_\bot)$ is *not* equivalent to $\texttt{if}(\texttt{e})\{\texttt{e}_\top\}\texttt{else}\{\texttt{e}_\bot\}$ since in the former *both* expressions $\texttt{e}_\top$ and $\texttt{e}_\bot$ are evaluated independently of the value of `e` (and thus there is no splitting into independent sub-networks). In order to enhance readability, in the remainder of this paper we shall avoid parentheses for nullary constructors (e.g. write `0` for `0()`); follow common infix notation and operator precedence for operators (e.g. write `x && !y` for `&&(x, !(y))`);

write `mux(e){e⊤}else{e⊥}` for `mux(e, e⊤, e⊥)`; and use syntax highlighting on snippets of field calculus code.

Example 1. As an example illustrating the constructs of field calculus, consider the problem – typical in sensor networks – of locally detecting dangerous situations in a working area and propagating alarms to all the devices in the same area. Assume `workingArea` is a Boolean built-in sensor identifying an area to be monitored (i.e., which is `true` in the devices within that very working area, and `false` elsewhere) and `danger` is a Boolean built-in sensor which holds `true` if some threat is detected. The goal is to build a Boolean field of alarms such that it becomes `true` in the `workingArea` whenever any device located there perceives some `danger`. The main expression uses construct **`if`** to limit the computation to the space-time region where `workingArea()` gives `true` (simply returning `false` elsewhere):

```
if (workingArea()) { gossipEver(danger()) } else { false }
```

We have made use of abstraction and specified the logic of alarm propagation through a function **`gossipEver`**, defined as follows:

```
def gossipEver(alarm) {
    share (false) { (old) => alarm || anyHoodPlusSelf(old) }
}
```

This function takes a Boolean field `alarm` and, whenever it holds a `true` value in a device, this gets propagated throughout the network by gossiping. This is achieved by using **`share`** to handle state and communication: a device is alarmed if it is currently perceiving some danger (`alarm` is true) or any of its neighbours, including itself, have perceived danger in their previous round (`anyHoodPlusSelf(old)`). Notice that when a device is alarmed, it continues to be so indefinitely (unless changes in the working area cause the computation to be skipped, hence refreshing the corresponding state).

A formal account of the field calculus operational semantics, formalising the behaviour of a firing, is given in Appendix A. Essentially, each expression evaluation creates a tree of "values" (corresponding to the unfolding of expression evaluation), containing values to be exported to neighbours (due to operators **`nbr`** and **`share`**): at the end of a firing, the resulting tree is packed and sent to neighbours, which will use it locally to support the semantics of **`nbr`** and **`share`**.

The computation within a single device is modeled by judgement "$\delta; \Theta; \sigma \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta$", to be read "expression $\mathtt{e}_{\mathtt{main}}$ evaluates to θ on device δ with respect to the value-tree environment Θ and sensor state σ", where θ is the evaluation result of $\mathtt{e}_{\mathtt{main}}$ as a value-tree, Θ is a map from each neighbour device δ_i (including δ itself) to the value-tree θ_i produced in its last firing, and σ is a data structure containing sensor information required to compute built-ins (like `workingArea` and `danger` of Example 1) related to sensors.

The overall evolution of the network is then modeled by a transition system $\xrightarrow{act}$ on network configurations $N = \langle Env; \Psi \rangle$. $Env = \langle \rightarrowtail, \Sigma \rangle$ models the environmental conditions, where the device neighbouring relation $\rightarrowtail$ (c.f. Definition 5) models network connection topology and the computational field Σ

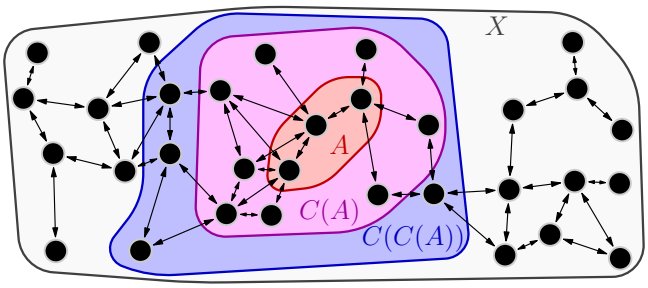


Figure 4: Pictorial representation of a closure space X induced by a graph.

(c.f. Definition 6) models sensor values on each device (see Appendix A.2 for further details). Ψ models the overall status of the devices in the network at a given time (as a map from device identifiers to value-tree environments Θ), and actions *act* can either be firings of a device (δ) or network configuration changes (*env*).

The system evolution formalised as a sequence of transition $\mathcal{S} = N_0 \xrightarrow{act_1} N_1 \xrightarrow{act_2} \dots$ can then be modeled through the more abstract notion of augmented event structure and space-time value (c.f. Section 2.2.1). In fact, every system evolution $\mathcal{S}$ induces a unique LUIC augmented event structure $\mathbf{E}$ (i.e., $\mathbf{E}$ is completely determined by $\mathcal{S}$), whereas every LUIC augmented event structure is induced by multiple system evolutions, as shown in the following theorem.

Theorem 1 (Semantic Completeness)**.** *Let* $\mathbf{E}$ *be a LUIC augmented event structure. Then there exist (infinitely many) system evolutions* $\mathcal{S}$ *that induce* $\mathbf{E}$*.*

Proof. See Appendix B.1. □

This result ensures that the linearity, uniqueness, impersistence, and computation immediacy constraints characterise exactly the set of event structures that can arise from the execution of such a system.

2.3. Spatial logics and SLCS

In traditional model checking and runtime verification of distributed and concurrent systems, properties expressed as a temporal logic formula (see for instance Ben-Ari et al. (1983); Emerson (1990)) are either statically or dynamically checked for satisfaction. Thus, properties of the temporal evolution of a system are considered, but properties of (physical) space typically are not. Spatial logics (van Benthem and Bezhanishvili, 2007) are topological interpretations of modal logics, whose purpose is to enable reasoning about the spatial dimension. These logics are based on two main modalities: $\Box\phi$ (holds in the interior of points where ϕ holds) and $\Diamond\phi$ (holds in the closure of points where ϕ holds). Other modalities have been considered during the years: of particular interest to us is a spatial surrounded operator—inspired by the temporal weak until operator—that first appeared for topological spaces in (Aiello, 2002).

In Ciancia et al. (2014), a Spatial Logic of Closure Spaces (SLCS in short) based on the above mentioned operators has been formalised for the more general setting of *closure spaces*, which can be formalised as a set X together with a closure operator $C : 2^X \to 2^X$ mapping set of points to their closure.[5] These spaces include the category of *quasi-discrete closure spaces*, which are conveniently characterised as the topologies arising from discrete directed graphs $\mathbf{G} = (D, \rightarrowtail)$ with $\rightarrowtail \subseteq D^2$, $\langle \delta, \delta \rangle \in \rightarrowtail$ for all $\delta \in D$ (c.f. network graphs as in Definition 5): in this case, the set of points is $X = D$ and closure is interpreted as "proximity": $C(A) = \{x \in D \mid \exists a \in A.\ x \rightarrowtail a\}$.[6]

A logic on quasi-discrete closure spaces is thus able to express properties of discrete networks of devices: a pictorial representation of this concept is given in Figure 4. In Ciancia et al. (2014), an efficient proof-of-concept model checker for SLCS on quasi-discrete closure spaces has been implemented.[7] In Section 3 we shall investigate how to perform distributed runtime verification on SLCS properties through aggregate computing techniques (in particular, devising a translation of properties into field calculus programs computing their truth in every node of the network of computing devices). In Ciancia et al. (2014), the presentation of the SLCS logic was first given in terms of the closure operator; then, for the special case of quasi-discrete closure spaces, formulations of SLCS operators in terms of paths in a graph were devised and proven equivalent. In the remainder of this paper, we shall only consider quasi-discrete closure spaces, modelled through network graphs, using the equivalent formulations of SLCS operators in terms of paths in a graph as their primitive definition.

Figure 5 (top) presents the full syntax of SLCS, comprising five "local" modalities and five "global" ones. The local modalities are:

- $\Box\, \phi$ (interior) which is true at points with all neighbours satisfying ϕ;[8]
- $\Diamond\, \phi$ (closure) which is true at points with some neighbour satisfying ϕ;
- $\partial\, \phi$ (boundary) which is true at points with some (but not all) neighbours satisfying ϕ;
- $\partial^-\, \phi$ (interior boundary) which is true at points satisfying ϕ with some neighbour not satisfying it;
- $\partial^+\, \phi$ (closure boundary) which is true at points not satisfying ϕ with some neighbour satisfying it.

The global modalities are:

- $\phi\, \mathcal{R}\, \psi$ (reaches) which is true at the ending points of paths (i.e., sequences $\delta_1 \rightarrowtail \ldots \rightarrowtail \delta_n$ in the directed graph $\mathbf{G}$ inducing the quasi-discrete closure space) whose starting point satisfy ψ and where ϕ holds;

[5] The closure operator has to satisfy $C(\emptyset) = \emptyset$, $A \subseteq C(A)$, and $C(A \cup B) = C(A) \cup C(B)$.
[6] Notice that $C(A) \supseteq A$ since $a \rightarrowtail a$ for each $a \in A$.
[7] Available at `https://github.com/vincenzoml/slcs`.
[8] Recall that every point is a neighbour of itself, since $\langle \delta, \delta \rangle \in \rightarrowtail$ for all $\delta \in D$.

$$\phi ::= \bot \mid \top \mid q \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \Rightarrow \phi) \mid (\phi \Leftrightarrow \phi) \quad \text{logical operators}$$
$$\mid (\Box\,\phi) \mid (\Diamond\,\phi) \mid (\partial\,\phi) \mid (\partial^{-}\,\phi) \mid (\partial^{+}\,\phi) \quad \text{spatial operators}$$
$$\mid (\phi\,\mathcal{R}\,\phi) \mid (\phi\,\mathcal{T}\,\phi) \mid (\phi\,\mathcal{U}\,\phi) \mid (\mathcal{G}\,\phi) \mid (\mathcal{F}\,\phi)$$

$$\begin{array}{ll}
\mathcal{M},\delta \models \top & \Leftrightarrow \mathit{true} \\
\mathcal{M},\delta \models q & \Leftrightarrow \delta \in \mathcal{V}(q) \\
\mathcal{M},\delta \models \neg\phi & \Leftrightarrow \mathcal{M},\delta \not\models \phi \\
\mathcal{M},\delta \models \phi \wedge \psi & \Leftrightarrow \mathcal{M},\delta \models \phi \ \mathit{and}\ \mathcal{M},\delta \models \psi \\
\mathcal{M},\delta \models \Diamond\,\phi & \Leftrightarrow \exists \delta' \rightarrowtail \delta.\ \mathcal{M},\delta' \models \phi \\
\mathcal{M},\delta \models \phi\,\mathcal{R}\,\psi & \Leftrightarrow \exists p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \mathcal{M},p_{|p|} \models \psi \ \mathit{and}\ \forall i \leq |p|\,.\ \mathcal{M},p_i \models \phi \\
\mathcal{M},\delta \models \phi\,\mathcal{T}\,\psi & \Leftrightarrow \exists p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \mathcal{M},p_{|p|} \models \psi \ \mathit{and}\ \forall i < |p|\,.\ \mathcal{M},p_i \models \phi \ \mathit{and}\ |p| \geq 2 \\
\mathcal{M},\delta \models \phi\,\mathcal{U}\,\psi & \Leftrightarrow \mathcal{M},\delta \models \phi \ \mathit{and}\ \forall p \in \mathcal{P}_{\mathbf{G}}(\delta)\ \mathit{if}\ \mathcal{M},p_{|p|} \not\models \phi\ \mathit{then}\ \exists i \leq |p|\,.\ \mathcal{M},p_i \models \psi \\
\mathcal{M},\delta \models \mathcal{G}\,\phi & \Leftrightarrow \forall p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \forall i \leq |p|\,.\ \mathcal{M},p_i \models \phi \\
\mathcal{M},\delta \models \mathcal{F}\,\phi & \Leftrightarrow \exists p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \exists i \leq |p|\,.\ \mathcal{M},p_i \models \phi
\end{array}$$

$$\begin{array}{rclrcl}
\Box\,\phi & \triangleq & \neg(\Diamond(\neg\phi)) & \partial\,\phi & \triangleq & (\Diamond\,\phi) \wedge \neg(\Box\,\phi) \\
\partial^{-}\,\phi & \triangleq & \phi \wedge \neg(\Box\,\phi) & \partial^{+}\,\phi & \triangleq & (\Diamond\,\phi) \wedge \neg\phi \\
\phi\,\mathcal{T}\,\psi & \triangleq & \phi\,\mathcal{R}(\Diamond\,\psi) & \phi\,\mathcal{U}\,\psi & \triangleq & \phi \wedge \Box\,\neg(\neg\psi\,\mathcal{R}\,\neg\phi) \\
\mathcal{F}\,\phi & \triangleq & \top\,\mathcal{R}\,\phi & \mathcal{G}\,\phi & \triangleq & \neg\,\mathcal{F}\,\neg\phi
\end{array}$$

Figure 5: Syntax (top) and semantics (centre) of SLCS; together with a reduction to a minimal set of modalities $\{\Diamond, \mathcal{R}\}$ (bottom).

- $\phi\,\mathcal{T}\,\psi$ (touches) which is true at the ending points of paths whose starting point satisfy ψ and where ϕ holds in the rest of the path;
- $\phi\,\mathcal{U}\,\psi$ (surrounded by) which is true at points in an area A satisfying ϕ, whose neighbours satisfy ψ;
- $\mathcal{G}\,\phi$ (everywhere) which is true at points where ϕ holds in every incoming path;
- $\mathcal{F}\,\phi$ (somewhere) which is true at points where ϕ holds in a point of an incoming path.

The whole list of modalities is redundant, meaning that they can all be expressed through $\Diamond$ and $\mathcal{R}$ only, by means of the equivalences expressed in Figure 5 (bottom). Figure 5 (centre) presents a semantics for a minimal set of logical connectives and local modalities, and for every global modality. Semantics models are of the form $\mathcal{M} = \langle \mathbf{G}, \mathcal{V} \rangle$, where $\mathbf{G} = \langle D, \rightarrowtail \rangle$ is a network graph and $\mathcal{V} : Q \to 2^D$ maps every propositional variable $q \in Q$ to the subset of devices where q holds. Semantics for global modalities is defined through properties of *paths in graph* $\mathbf{G}$ *towards* δ, defined as sequences $p = \langle \delta_1, \ldots, \delta_n \rangle$ such that $\delta_n \rightarrowtail \ldots \rightarrowtail \delta_1 = \delta$. We use $\mathcal{P}_{\mathbf{G}}(\delta)$ to denote the set of such paths.

SLCS being a spatial logics, it is worth noticing that its formulas generally have different values in different points of space. A notable exception is the case of formulas that are a logical combination (i.e., via non-modal operators)

of formulas that have $\mathcal{G}$ or $\mathcal{F}$ as top connective, in a strongly connected[9] graph: in these cases they are either true or false in all points of space—so, the specified monitors emit a global verdict.

Remark 2 (On the Relation with Ciancia et al. (2014)). The presentation of SLCS in Ciancia et al. (2014) differs in many relevant though not fundamental ways.

Firstly, semantics was primitively given for closure spaces, substituting $\mathbf{G}$ with pairs $\langle X, C\rangle$ where $C : 2^X \to 2^X$, and only afterwards a semantic interpretation on graphs through paths was derived (Ciancia et al., 2014, Theorem 3 and Remark 3). Even though the semantics on closure spaces is more general, it is not exploited by the model checker in Ciancia et al. (2014), which only applies to quasi-discrete closure spaces defined in terms of graphs. Since the generality of closure spaces does not seem to translate into additional applicability, we opted for a presentation natively rooted on graphs.

Secondly, the primitive global modality in Ciancia et al. (2014) is $\mathcal{U}$ instead of $\mathcal{R}$, since $\mathcal{U}$ has a cleaner definition in terms of closure spaces: $\phi \,\mathcal{U}\, \psi$ holds in x iff $\exists A$ such that $x \in A$, ϕ holds in A and ψ holds in $\partial^+ A$. Conversely, in our presentation rooted on graphs $\mathcal{R}$ has a simpler definition than $\mathcal{U}$ (and is more easily computable), and thus was chosen as a primitive modality.

Thirdly, and most importantly, the operator $\tilde{\mathcal{R}}$ called *reach* in Ciancia et al. (2014) is similar but not identical to ours. There, $\mathcal{M}, \delta \models \phi \,\tilde{\mathcal{R}}\, \psi$ if and only if:

$$\exists p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \mathcal{M}, p_{|p|} \models \psi \ \textit{and}\ \forall i = 2, \ldots, |p| \,.\ \mathcal{M}, p_i \models \phi$$

In particular, the difference is in that ϕ is *not* required to hold in δ. Although this choice being counter-intuitive (and inconvenient in practice), this version of the reachability operator was preferred since it is the dual of the $\mathcal{U}$ operator: $\phi \,\tilde{\mathcal{R}}\, \psi \triangleq \neg(\neg\psi \,\mathcal{U}\, \neg\phi)$. We opted for a presentation tailored for formulas on graphs, for which operator $\mathcal{R}$ is more relevant (and easily computable) than $\tilde{\mathcal{R}}$. Notice also that the two operators are interchangeable through the equivalences $\phi \,\mathcal{R}\, \psi = \phi \wedge (\phi \,\tilde{\mathcal{R}}\, \psi)$ and $\phi \,\tilde{\mathcal{R}}\, \psi = \psi \vee \Diamond(\phi \,\mathcal{R}\, \psi)$.

Finally, the *touch* operator is not present in Ciancia et al. (2014). However, it is defined in Ciancia et al. (2018) as $\phi \,\mathcal{T}\, \psi \triangleq \phi \wedge \Big((\phi \vee \psi) \,\tilde{\mathcal{R}}\, \psi\Big)$, which can be proven equivalent to our definition.

Remark 3 (On Modality Equivalences). Since we both gave a direct semantic interpretation of $\mathcal{T}$, $\mathcal{U}$, $\mathcal{G}$, $\mathcal{F}$ and a definition of them in terms of $\mathcal{R}$, the two must be proven equivalent. For $\mathcal{G}$, it boils down to an easy substitution exercise. For $\mathcal{F}$, substitution gives $\exists p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \mathcal{M}, p_{|p|} \models \phi$; and if a path satisfies the direct interpretation of $\mathcal{F}$, the restricted path $p' = \langle p_1, \ldots, p_i\rangle$ satisfies the interpretation through $\mathcal{R}$. For $\mathcal{T}$, substitution gives the following:

$$\exists p \in \mathcal{P}_{\mathbf{G}}(\delta).\ \exists \delta' \rightarrowtail p_{|p|}.\ \mathcal{M}, \delta' \models \psi \ \textit{and}\ \forall i \leq |p| \,.\ \mathcal{M}, p_i \models \phi$$

[9] A directed graph $\mathbf{G}$ is strongly connected iff for every two points δ_1, δ_2 in $\mathbf{G}$ there exists a path from δ_1 to δ_2 in $\mathbf{G}$.

which is satisfied whenever the direct interpretation of $\mathcal{T}$ is satisfied with the extended path $p' = \langle p_1, \ldots, p_n, \delta' \rangle$.

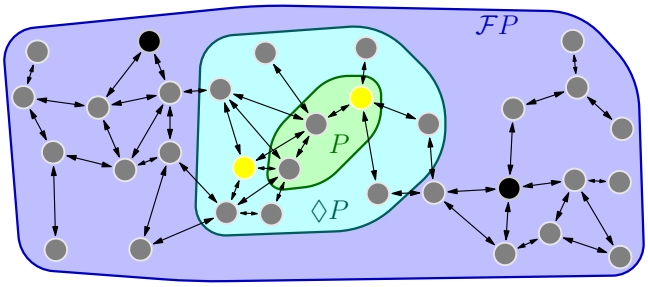
For $\mathcal{U}$, we use the result from Ciancia et al. (2014) that $\phi \,\mathcal{U}\, \psi \;=\; \neg(\neg\psi \,\tilde{\mathcal{R}}\, \neg\phi)$ (with the semantic interpretation of $\mathcal{U}$ in Figure 5 (centre) and the semantic interpretation of $\tilde{\mathcal{R}}$ in Remark 2). We can then use the equivalence $\phi \,\tilde{\mathcal{R}}\, \psi \;=\; \psi \vee \Diamond(\phi \,\mathcal{R}\, \psi)$ in order to obtain that:

$$\phi \,\mathcal{U}\, \psi = \neg\left(\neg\phi \vee \Diamond(\neg\psi \,\mathcal{R}\, \neg\phi)\right) = \phi \wedge \Box\, \neg(\neg\psi \,\mathcal{R}\, \neg\phi).$$

Example 2 (Smart Home)**.** As sample application of SLCS in a smart home scenario, consider the following property to monitor: *air conditioning and lights are on whenever the room is not empty, off otherwise.* Consider the atomic propositions:

- P is true on points that are sensing the presence of people;
- D is true on points that are the monitored electrical devices (air conditioning, lights);
- O is true on electrical devices that are on.

If we only want to consider the presence of people in the immediate vicinity, the considered property can be written as $\neg D \vee (O \Leftrightarrow \Diamond\, P)$. When also considering people farther away, the property can be written as $\neg D \vee (O \Leftrightarrow \mathcal{F}\, P)$. In the sample closure space of Figure 4, a possible evaluation of these properties could be the following, where different colours are used for points where D is false (grey), D is true and O is false (black), D and O are true (yellow):



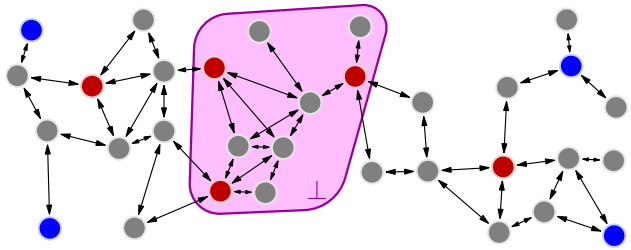
The green area denotes the nodes for which P is true, i.e., the nodes that do perceive at least a person in the room. The cyan area ($\Diamond\, P$) is given by the nodes that have at least a neighbour within the green area. The blue area ($\mathcal{F}\, P$) is given by the nodes for which there exists a path to a node in P ("somewhere"); this includes all the nodes of the example.

Example 3 (Sensor Network)**.** As sample application of SLCS in a sensor network scenario, consider the following property to monitor: *internet is reachable through non-busy devices.* Consider the atomic propositions:

- B is true on busy devices;

- I is true on devices that have an internet connection.

The considered property can then be written as $\neg B \,\mathcal{R}\, I$. An evaluation of this formula is represented in the following picture, where the purple area marks points where the formula is false and different colours are used for points where B is true (red), I is true (blue), or none are true (grey).

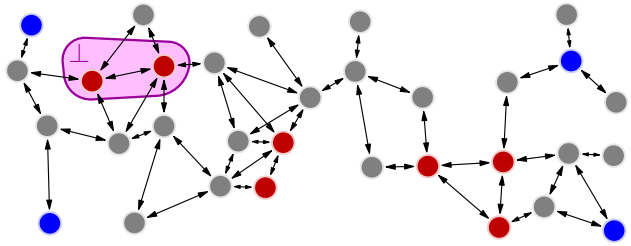


In the central part of the network, the property is false because there is no path from the grey nodes inside it to a blue (Internet) node which does not pass through a red (busy) node; i.e., the red nodes make up a perimeter which does not contain any blue node.

Example 4 (Emergency)**.** As sample application of SLCS in an emergency scenario, consider the following property to monitor: *dangerous areas are surrounded by non-dangerous areas from which it is possible to reach a recovery point without crossing any other dangerous area.* Consider the atomic propositions:

- D is true on devices in dangerous areas;
- R is true on devices in recovery points.

The considered property can then be written as $D \Rightarrow (D \,\mathcal{U}(\neg D \,\mathcal{R}\, R))$. An evaluation of this formula is represented in the following picture, where the purple area marks points where the formula is false and different colours are used for points where D is true (red), R is true (blue), or none are true (grey).



The property is false for the two red (dangerous area) nodes at the top left of the network because the grey node above them is unable to reach a blue (recovery point) node without passing through a red (dangerous) node.

$[\![\top]\!] = \texttt{true}$	$[\![\phi_1 \vee \phi_2]\!] = [\![\phi_1]\!] \texttt{ || } [\![\phi_2]\!]$
$[\![\bot]\!] = \texttt{false}$	$[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \texttt{ \&\& } [\![\phi_2]\!]$
$[\![q]\!] = \texttt{q()}$	$[\![\phi_1 \Rightarrow \phi_2]\!] = [\![\phi_1]\!] \texttt{ <= } [\![\phi_2]\!]$
$[\![\neg\phi]\!] = \texttt{!}[\![\phi]\!]$	$[\![\phi_1 \Leftrightarrow \phi_2]\!] = [\![\phi_1]\!] \texttt{ == } [\![\phi_2]\!]$
$[\![\Diamond\,\phi]\!] = \texttt{anyHoodPlusSelf(nbr\{}[\![\phi]\!]\texttt{\})}$	$[\![\phi_1\,\mathcal{R}\,\phi_2]\!] = \texttt{reaches(}[\![\phi_1]\!]\texttt{, }[\![\phi_2]\!]\texttt{)}$
$[\![\Box\,\phi]\!] = \texttt{allHoodPlusSelf(nbr\{}[\![\phi]\!]\texttt{\})}$	$[\![\mathcal{F}\,\phi]\!] = \texttt{somewhere(}[\![\phi]\!]\texttt{)}$

Figure 6: Translation $[\![\phi]\!]$ of an SLCS formula ϕ into field calculus. Only the logical operators and the spatial operators $\Box$, $\Diamond$, $\mathcal{F}$ and $\mathcal{R}$ are considered—the other spatial operators (∂, ∂^-, ∂^+, $\mathcal{T}$ and $\mathcal{U}$) can be translated by rewriting them through the considered ones according to Figure 5 (bottom).

```
def distanceTo(dest) {
  share (infinity){ (d) => mux (dest) {0} else {minHood(d)+1} }
}
def somewhere(x) {
  distanceTo(x) <= D
}
def reaches(x, y) {
  if (x) {somewhere(y)} else {false}
}
```

Figure 7: Functions **somewhere** and **reaches** used in the translation of Figure 6, in turn using the auxiliary function **distanceTo**.

3. Automatic generation of distributed monitors in field calculus

In this section, we provide a translation of SLCS formulas into field calculus. Namely, thanks to the functional nature of field calculus, the resulting distributed monitor will provide efficient computation of the truth value of a formula at each device by recursion over its syntactic structure, and this will be achieved assuming that each participant is evaluating the same property from its perspective with regard to any quantifiers. In particular, we translate atomic propositions q into built-in function calls getting their value from some external environment (which we do not detail here, since typically involving external sensors), logical connectives into corresponding Boolean built-ins, and modal spatial operators into field calculus library functions that perform spatial operations (such as propagating values through spanning trees to compute distances). The functional composition character of field calculus, which works at the global level, is the distinctive feature by which this model allows to easily express the translation, as well as to formally prove self-stabilisation of the resulting monitors. Section 3.1 introduces a translation of SLCS into field calculus, coherently with the formal interpretation (c.f. Section 2.3). Section 3.2 discusses the correctness and efficiency properties of this translation.

3.1. Automatic translation in field calculus

Figures 6 and 7 show the translation $[\![\phi]\!]$ of an SLCS formula ϕ into field calculus, which essentially derives a space-time function that turns Boolean

space-time values for the atomic propositions (q) into a Boolean space-time value representing the validity of the monitored formula over space and time—validity depends on time in the transient phase, when the validity of atomic propositions symbols change, or if network topology changes. The correspondence of the translation with the SLCS semantics (c.f. Figure 5) is formally given by Theorem 3, which shows that the limit of the translated space-time value always match the computational field defined by the SLCS formula (c.f. Definition 8). This translation uses the data constructors and built-in functions described in Figure 3, and assumes that:

- `false < true`, as in common programming languages such as Python;
- **`distanceTo`** is a commonly used algorithm (based on Bellman-Ford algorithm, available in the standard libraries of field calculus implementations) determining the shortest distance (in network hops) to a destination point (Viroli et al., 2018);
- **somewhere** is a field calculus function which is true whenever the shortest distance to a point where the argument holds is plausible (smaller than the network diameter `D` in hops. In many cases, the parameter D can be determined at network design time, allowing to run the translation in Figure 6 as is. When this is not possible, the translation can still be applied by incorporating strategies for dynamically estimating the diameter: e.g., computing the maximum distance from an elected leader (Mo et al., 2020);
- **`reaches`**`(x, y)` is a field calculus function which is true only in connected areas where `x` is true and somewhere in that area `y` is also true (since in that case there must be a path staying within the area where `x` is true which reaches a point where `y` is true).

Remark 4 (Translation of Derived Modalities). The translation in Figure 6 is defined also for the derived modalities $\Box$, $\mathcal{F}$ and all derived logical operators. The translation given is coherent with the definition of those derived operators in terms of the primitive ones. For logical operators, this equivalence is standard. For the *interior* modality, notice that its definition $\Box\,\phi \triangleq \neg(\Diamond(\neg\phi))$ in terms of primitives is translated into the (valid) equivalence:

$$\texttt{allHoodPlusSelf(}[\![\phi]\!]\texttt{)} \;\equiv\; \texttt{!anyHoodPlusSelf(!}[\![\phi]\!]\texttt{).}$$

Similarly, the definition $\mathcal{F}\,\phi \triangleq \top\,\mathcal{R}\,\phi$ of $\mathcal{F}$ in terms of primitives translates to the valid equivalence:

$$\begin{aligned} \textbf{somewhere}\texttt{(}[\![\phi]\!]\texttt{)} \quad &\equiv \quad \textbf{if}\ \texttt{(true)}\ \texttt{\{}\textbf{somewhere}\texttt{(}[\![\phi]\!]\texttt{)\}}\ \textbf{else}\ \texttt{\{false\}} \\ &\equiv \quad \textbf{reaches}\texttt{(true, }[\![\phi]\!]\texttt{)} \end{aligned}$$

In the next section, the coherence between translation and operator definitions explained in Remark 4 will be extended by (and be a consequence of) Theorem 3, which proves that the translation is coherent with the semantics of all primitive and derived operators (c.f. Figure 5 centre).

It may be tempting to implement **somewhere** with function **gossipEver** in Example 1, instead of the implementation proposed in Figure 6. However, this function is not able to adjust its value from `true` to `false` in case all the points satisfying the argument disappear from the network (namely, it is not self-stabilising c.f. Definition 10): thus, this approach would only work in a fully static situation and could not be used in a dynamic environment for tracking the truth value of the spatial formula over time. Several approaches could be used for implementing a self-adjusting **somewhere** routine in field calculus: e.g. with replicated gossip (Pianini et al., 2016) or by combining several commonly used building blocks (Beal and Viroli, 2014) (S for leader election, G for distance estimation and broadcasting, C for data summarisation). The proposed one, however, excels by its simplicity and efficiency (see Theorem 2) while providing optimal reactiveness to input changes (see Theorem 3).

The reader may appreciate the simplicity of the proposed translation, which is compositional (i.e. defining the translation of an expression by composing translations of sub-expressions) thanks to the functional paradigm of field computations, and which hints at the power of field calculus as an implementation technique for higher-level languages and logic frameworks. Indeed, complex behaviours can arise at the level of device interactions, but they are hidden under the hood of the computational model and the lower-level functions used in the translation.

Example 5 (Formula Translation)**.** The translation of the surrounded by operator $\phi_1 \mathcal{U} \phi_2$ is **surroundedBy**$(\![\phi_1]\!], [\![\phi_2]\!])$, where

```
def surroundedBy(x, y){
  x && allHoodPlusSelf(nbr{!reaches(!y,!x)})
}
```

Then, the translation of the formulas in Examples 2, 3, 4 are the following.

$\neg D \vee (O \Leftrightarrow \Diamond P)$	`!D() || (O() == anyHoodPlusSelf(nbr{P()}))`
$\neg D \vee (O \Leftrightarrow \mathcal{F} P)$	`!D() || (O() == somewhere(P()))`
$\neg B \mathcal{R} I$	`reaches(!B(), I())`
$D \Rightarrow (D \mathcal{U} (\neg D \mathcal{R} R))$	`D() <= surroundedBy(D(), reaches(!D(), R()))`

3.2. Properties of the translation

Firstly, we show that the monitors obtained from the translation of SLCS formulas are efficient and lightweight, being able to scale to arbitrary network sizes and easily runnable on low-end devices.

Theorem 2 (Lightweightness)**.** *The translation* `P` *of a formula* ϕ *according to Figure 6 computes in each node using message size* $O(S)$ *and computation time/space* $O(L + SN)$*, where* N *is neighbourhood size and* L, S *are the numbers of logical and spatial operators in* ϕ*.*

Proof. See Appendix B.2. □

Notice that the bounds on message size, computation time and space provided above are asymptotically optimal, and thus the translated program can

be deemed efficient and lightweight. Furthermore, these bounds do not depend on the network size, implying that the computation can scale up to arbitrarily large networks, as long as the individual degree of nodes stays bounded. In practice, we can expect a firing to be executed within few microseconds of CPU time in any realistic scenario.

We are now able to prove that the given translation is correct and optimal. Correctness will be shown in terms of *stabilisation* to the correct *limit*, as formalised in Definition 8 for space-time functions. In fact, this definition can be translated to field calculus functions and expressions by means of the following definition:

Definition 11 (Stabilising Expression)**.** A field calculus expression `e` is *stabilising* with limit Ψ on $\mathbf{G}$ iff for any system evolution $\mathcal{S} = N_0 \xrightarrow{act_1} \dots$ of program `e` such that for some n_0 and each $n \geq n_0$, the environment Env_n in N_n is the same[10] and has topology given by $\mathbf{G}$, and there are infinitely many δ transitions for each δ in $\mathbf{G}$, then for some n_1 and each $n \geq n_1$ the value produced in each firing δ is exactly $\Psi(\delta)$.

Optimality will then be measured as having the lowest bound on the number of *full rounds of execution* required for stabilisation.

Definition 12 (Full round of execution)**.** Let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \dots$ be a (possibly infinite) network evolution consisting only of device fires. We say that one *full rounds of execution* has passed at step t if t is minimal such that for each device δ in the network, there exists an $i < t$ such that $\delta_i = \delta$. Similarly, we say that n full rounds of execution has passed at step t if $n-1$ full rounds of execution has passed at step t', and one full round of execution has passed at step $t - t'$ in the reduced network evolution $N_{t'} \xrightarrow{\delta_{t'}} N_{t'+1} \xrightarrow{\delta_{t'+1}} \dots$.

Self-stabilisation of the monitor comes as a direct consequence of the results in Viroli et al. (2018), since all expressions in Figure 6 belong to the *self-stabilising fragment* of the field calculus thereby identified. In the following theorem, however, we shall also prove that its limit is the value of the SLCS formula (c.f. Figure 5 centre), and that it is obtained with the lowest possible number of full rounds of execution.

Theorem 3 (Self-Stabilisation, Correctness, Optimality)**.** *Let* `P` *be the translation of* ϕ *according to Figure 6. If the network configuration and atomic propositions stabilise, the result of* `P` *also stabilises to the interpretation of* ϕ *in that final configuration (regardless of the evolution history of the network). Furthermore, the time required for stabilisation is as small as possible, meaning that no correct inductive translation can stabilise with a smaller worst case of full rounds of execution.*

Proof. See Appendix B.3. □

[10] Notice that such a system has no *env* transitions after n_0.

According to the above theorem, and assuming the network is connected, then:

- When the network configuration and atomic propositions become stable, the field calculus monitor `P` stabilises (as fast as possible), and compute in each device the correct value of the SLCS formula ϕ.
- When the network configuration and atomic propositions keep evolving, the field calculus monitor `P` keeps bringing about the correct validity of the formula ϕ at each device: due to intrinsic delays in communication, then, the snapshot of a result at a device may not be correct. However, given the simple and natural structure of the **`distanceTo`** algorithm used in the translation, one still expects fast reactiveness to changes in a regime of persistent perturbation without stable points—even though it is harder to characterise reactiveness formally. This claim will be validated through simulation in Section 4.

Note that, if the network is divided in two or more disconnect sub-network, then the above bullets apply to each sub-network.

4. Case study: crowd safety

The last section proved the correctness and self-stabilising nature of field calculus translations of SLCS properties. Though field calculus monitors are guaranteed to *eventually* converge to the correct value, it is still open the question of whether they are reasonably reactive, i.e., whether the approach can be useful in application settings characterised by frequent or continuous change. Accordingly, in this section, we demonstrate the proposed approach in a large-scale computing scenario related to crowd safety (described in Section 4.1), by means of simulation (as described in Section 4.2). That is, we show a concrete example of translation from SLCS to field calculus, leading to a distributed monitor whose reactive behaviour is verified by means of repeated experiments. Most specifically, we focus on the ability of such a monitor to produce an output that eventually converges to that of an ideal monitor, implemented as an *oracle* that checks the SLCS property exhaustively in every state of the simulated system and accordingly provides, at each simulation step, the correct value that should be computed by every device participating in the system. Especially, we will show the ability of the monitor to keep up with continuous change (change in connection topology) in the environment. Indeed, while the oracle has instantaneous access to the global state of the distributed system under simulation, the field calculus monitor runs in a decentralised fashion, where each device can only directly observe (by exchanging messages with neighbours) a local portion of the overall system, and therefore it takes some time to converge to the correct value. Notice that a single local change in the system – e.g., a single connectivity link that changes – may potentially cause a SLCS property to globally flip (as exemplified in Figure 10). In other words, given an execution round of some device, the oracle provides the correct value that such a device could compute

if it had access to the global state of the evolving distributed system (which is clearly an unrealistic assumption). Moreover, though convergence is *eventually* guaranteed for constant input (as covered in Section 3), the monitor should also be "enough reactive" to bring the error (measured by the difference between the SLCS property field and the field provided by the oracle) at acceptable levels (which are, still, generally application-specific). Experimental results are presented in Section 4.3, whereas further discussion about this latter aspect is available in Section 4.4.

The source code, build infrastructure, and instructions for running and reproducing the experiments are publicly available online[11].

4.1. Simulation scenario

The scenario leverages real-world data of a recent mass event (Anzengruber et al., 2013), consisting of anonymised GPS traces recorded from a subset of the visitors. This is an example where very large numbers of people move around the city, possibly leading to bloats or situations of danger. In this setting, scalable crowd analysis and management algorithms can help to provide safety and services for a better experience, e.g., by estimating the density of the crowd, propagating information about the crowd to the surroundings, and supporting crowd-aware dispersal and navigation.

Aggregate programming techniques have proven to be effective in expressing such crowd management algorithms by a global perspective, and to make them execute in an adaptive, resilient and decentralised fashion (Beal et al., 2015; Casadei et al., 2019a). With this approach, each participant has a smartphone or another wearable device that provides sensor data (e.g., presence of people nearby) and outputs local information (e.g., the suggested direction of movement to avoid overcrowded areas) for the person as computed by the collectively executed aggregate program. In practice, multiple concrete deployments are possible, ranging from fully peer-to-peer to cloud-based architectural styles (Viroli et al., 2016). Given the costs of testing new services for the proposed real-world scenario, we proceed empirically by simulation, considering a reasonable aggregate system deployment where nodes perform their firing in a non-synchronous fashion and interact with all the devices closer to them with respect to a certain, configurable threshold. More details are provided in the next section and are available at the attached repository. Finally, it is key to consider that this approach does not require either global knowledge (e.g., the GPS positions of devices and locations) or global connectivity to the Internet (which may be limited in mass events), therefore providing a viable solution for guaranteeing (continuity of) services in situations where only minimal assumptions hold, through graceful degradation (if any).

For the purpose of this paper, we focus on monitoring the safety property represented by the SLCS formula in Example 4:

$$D \implies (D\,\mathcal{U}(\neg D\,\mathcal{R}\,B))$$

[11] https://github.com/metaphori/experiments-2019-ac-slcs-monitor-vienna

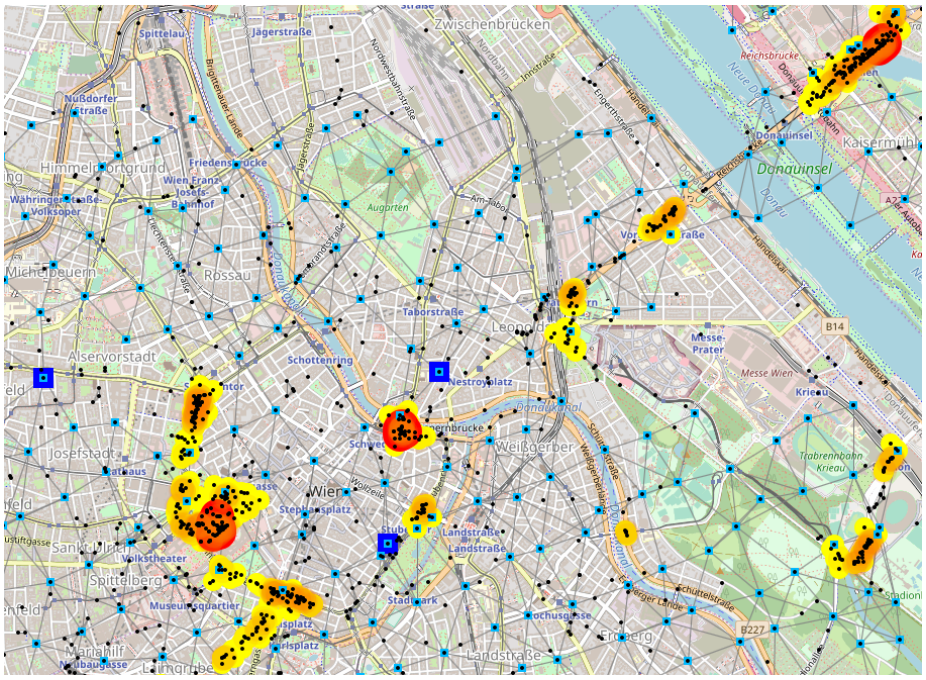


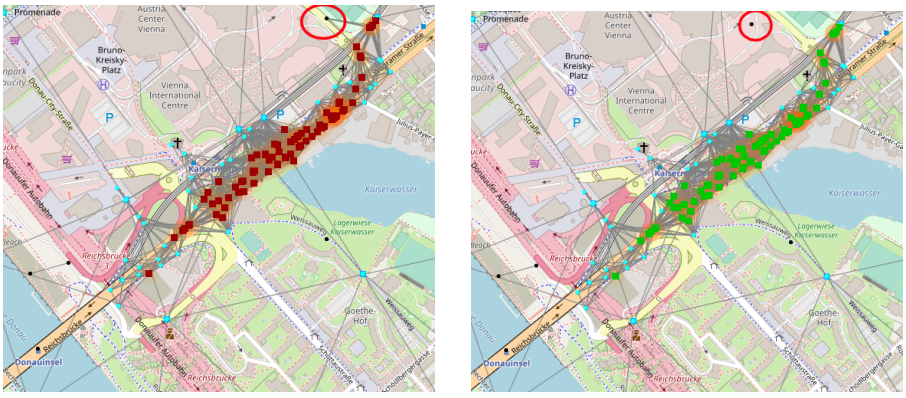
Figure 8: Representation of a snapshot of the simulated scenario, as a network of devices in the city: black dots denote (the smartphones of) people corresponding to the GPS traces of the reference mass event; grey links represent connectivity (i.e., the neighbouring relationships); yellow, orange, and red overlays represent increasing levels of crowding; blue squares denote safe places (these are real locations of hospital facilities); small, light blue squares represent access points.

where D denotes a "dangerous area" (i.e., an area which is overcrowded or in the very proximity of one) and B denotes a "base" or "safe area" (e.g., an exit for dispersal or an area with medical facilities). The above property can be read as "dangerous areas are surrounded by devices which can safely reach a base"; in other words, this is to guarantee that large groups of people do not hinder the way to exits or other important locations to other people. A visual representation of the scenario is provided in Figure 8 (full-size, colour pictures are included in the provided repository).

Note that the reachability of safe points is based on connectivity paths across the system of devices; in other words, each neighbouring link between two nodes should represent a valid hint for an accessible, walkable path. Therefore, in this simulated scenario it is assumed that links can actually be followed on foot, and that there is a sufficiently dense (but still quite sparse) network in place—otherwise, the system would consist of multiple isolated sub-networks with trivial results. To improve realism, for the simulations, the GPS traces are interpolated to place nodes on actual streets. Moreover, the system neglects other potential paths passing through streets which are not "sampled" by any

```
/* CROWD DETECTION FUNCTIONS */

def countNearby(range) {
  let human = rep(h <- env.get("role")==0) { h };
  sumHood(mux(human && nbrRange() < range) { 1 } else { 0 })
}

def densityEstimation(p, range, w) {
  countNearby(range) / (p * PI * range ^ 2 * w)
}

def dangerousDensity(p, range, densityTh, groupSize, w) {
  let partition = S(range, nbrRange);
  let localDensity = densityEstimation(p, range, w);
  let avg = summarize(partition, sum, localDensity, 0) /
            summarize(partition, sum, 1, 0);
  let count = summarize(partition, sum, 1 / p, 0);
  avg > densityTh && count > groupSize
}

def crowdTracking(p, range, w, density,
                  dangerousTh, groupSize, timeFrame) {
  if (isRecentEvent(densityEstimation(p, range, w) > density,
                    timeFrame)) {
    if (dangerousDensity(p, range, dangerousTh, groupSize, w)){
      OVERCROWDED
    } else { AT_RISK }
  } else { NONE }
}

/* SLCS FUNCTIONS */

def interior(f){ allHoodPlusSelf(nbr(f)) }
def closure(f){ anyHoodPlusSelf(nbr(f)) }
def somewhere(f){ hopDistanceTo(f) < DIAMETER }
def reaches(f1,f2) { if(f1){ somewhere(f2) } else { false } }
def surroundedBy(f1,f2){ f1 && interior(!reaches(!f2,!f1)) }
def implies(f1,f2) { f1 <= f2 }

/* PROGRAM: CROWD ESTIMATION */

let p = 0.005;    let w = 0.25;    let crowdRange = 30;
let crowdedDensity = 1.08;         let dangerousThreshold = 2.17;
let groupSize = 300;               let timeFrame = 60;
let crowding = crowdTracking(p, crowdRange, w, crowdedDensity,
                    dangerousThreshold, groupSize, timeFrame)

/* PROGRAM: PROPERTY TO BE MONITORED */

let D = crowding == OVERCROWDED || crowding == AT_RISK;
let B = env.get("isSafePlace");
implies(D, surroundedBy(D, reaches(!D, B)))
```

Figure 9: Protelis implementation of the aggregate specification executed for the case study. Bold red symbols denote language keywords; bold blue symbols denote standard library functions; bold purple symbols denote application-related functions; bold black symbols denote SLCS functions; and bold orange symbols denote parameters, constants, or built-ins whose declaration is not reported.



(a) The red-circled node has every path to a safe node hindered by a dangerous, crowded area—which is red-coloured to denote its collective failure in satisfying the property. The cyan circles denote nodes able to reach safety.

(b) The red-circled node walks away, detaching from the network. All the remaining nodes can reach a safe area (not shown) by passing across the bridge: therefore, the crowded area satisfies the property.

Figure 10: These zoomed snapshots of the scenario are meant to illustrate the property checking. We assume safe areas are only south of the river, and there are no paths that circumvent the Reichsbrücke bridge shown in the picture.

device; nevertheless, the presented solution also shall work when devices do not have maps (which, anyway, should be augmented with crowding data), since a "map" is implicitly constructed (though, of course, actual accessibility should be fostered with proper design decisions).

4.2. Simulation framework and setup

For these experiments, we leverage the meta-simulator Alchemist (Pianini et al., 2013) which provides an event-driven simulation engine for scheduling events and actions upon modelled entities as well as features for configuration of scenarios, visualisation and data extraction. The translated monitors are written in Protelis (Pianini et al., 2015), an implementation of the field calculus as a standalone domain-specific language that also provides a library of reusable aggregate building blocks and interoperability with the Java ecosystem.

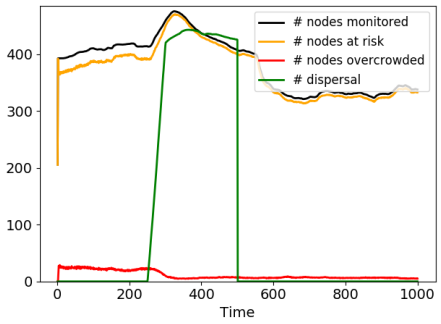
The scenario is configured as follows: a total of 1497 nodes are loaded at the starting positions of the corresponding available GPS traces and configured to move according to their traces as well as to execute the aggregate program and broadcast data to neighbours once about every $T_R = 1$ second(s). Since the network inferred by the data traces is quite sparse relatively to the physical region of the city, a mesh of access points is put in place to provide a reasonable level of connectivity for the system. While normal devices are assumed to have a connectivity range of $100m$ (i.e., around the maximum Wi-Fi range), access points connect to each other within a $500m$ range; these choices have been made as a compromise between simplicity and realism—more deployment-related considerations will follow.

An excerpt of the aggregate program implementation is provided in Figure 9. Dangerous areas (i.e., nodes where field `D` is true) are those which are either `OVERCROWDED` or `AT_RISK` (i.e., nearby overcrowded areas by a certain threshold), as computed by aggregate function `crowdTracking`, whereas safe areas are given by predefined nodes having a corresponding property as true. The crowd detection functionality and its parameterisation are taken from Beal et al. (2015). The last expression of the listing is the property to be monitored, expressed straightforwardly through the SLCS functions: it yields a Boolean field that is true in the nodes in which the property is satisfied, and false where this is not the case.

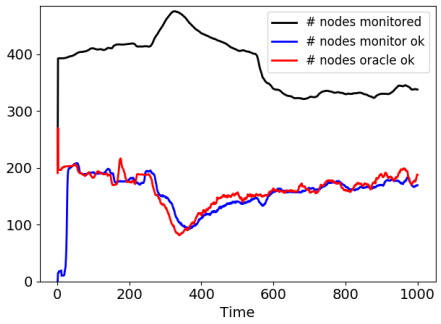
The code of Figure 9 is then extended with simulation-specific code for the parameterised configuration of the scenario, for the oracle, and for data extraction. The oracle, whose goal is to objectively check the property by inspecting the state of the system, is executed by-need when a device in D fires; its implementation leverages JGraphT (Michail et al., 2020) for representing the partitioned network of non/dangerous devices and calculating connected sets in order to check graph reachability of safe nodes. Moreover, in order to stress the monitor, there is additional aggregate code to activate, from $t = 250$ to $t = 500$, a simple crowd dispersal algorithm that changes network topology: nodes close to risk are suggested to move in the direction opposite to the nearest dangerous node (which performs a spatial broadcast of its GPS position). Nodes are configured with a certain, individual probability to follow the dispersal advice; so, when the advice is given, and they "choose" to follow it, they move in the suggested dispersal direction, hence departing from the recorded GPS trace.

4.3. Simulation results

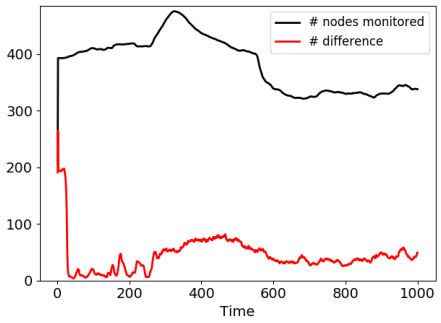
The scenario described above is run 100 times, each with a different random seed, yielding 100 different simulation instances. The random seed is given as input to the pseudo-random generator of the Alchemist simulator, which affects *(i)* the displacement of access points into random mesh-like arrangements (i.e., whereas the smartphones are positioned according to the GPS traces, which are always the same, and hospitals have precise fixed locations, the access points are positioned along a grid with random shifts along the ideal latitude and longitude positions in each different instance), *(ii)* the tendency of people to follow the dispersal advice, and *(iii)* the relative ordering of the computation rounds scheduled at the devices. These aspects may affect the dynamics of the system as well as the trajectory of the SLCS property in non-trivial ways, since the appearance or the vanishing of few connectivity links could make a huge difference. Each simulation instance is executed for 1000 seconds of simulated time. For each run, on every second of simulated time, the following data is exported: the number of devices which are overcrowded, at risk, and monitored (the monitor needs to run on all the devices, but the property is relevant only on devices in D, so, as an optimisation, only for these the oracle is executed); the number of devices for which the monitor and the oracle yield a positive response, as well as the number of devices for which the monitor and the oracle provide a different response (i.e., this is a measure of the error), detailed with



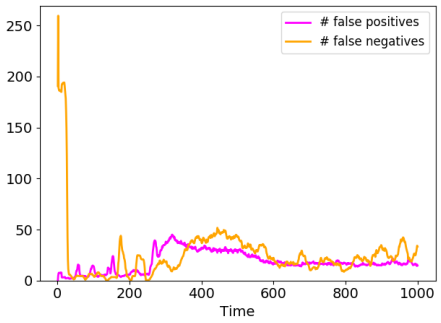
(a) Number of devices that perceive a dangerous (red line) or moderate (orange line) overcrowding, monitored (black line), and following dispersal advices (green line).



(b) Number of monitored devices (black line) that satisfy the property according to the oracle (red line) and the monitor (blue line).



(c) Number of monitored devices (black line) for which the oracle and the monitor provide a different response (red line).



(d) Detail of the error in terms of the number of devices providing false positives (magenta line) and false negatives (orange line).

Figure 11: Simulation results.

the count of false positives (i.e., erroneous monitor evaluations suggesting the property is satisfied when it is not the case) and false negatives (i.e., erroneous unsatisfiability claims).

The results are shown in Figure 11. In particular, Figure 11a shows how the level of crowding varies over time, also by the effect of the crowd dispersal process, which makes nodes in danger and those close to risky areas disperse. It is possible to appreciate a reduction of the dangerous areas, as well as a reduction of devices at risk (after an initial increase due to the dispersal dynamics).

In Figure 11b, we have a view of how the property evolves in the system and, more interestingly, we can observe how the response of the distributed, field calculus monitor "follows" the response of the oracle, with a certain delay and with a varying error that can be more precisely observed in Figure 11c (by counting the number of different individual responses) and in Figure 11d (by counting the number of false positives and false negatives). Moreover, after the crowd dispersal, the number of devices for which the property is true is larger *relative to* the number of monitored devices. The error for this scenario is, on average, around 10-20%; though not bad, this figure could easily improve by increasing system stability (the sparseness of the used dataset produces high variability due to network fragmentation). Also, interestingly, note that the monitor tends to provide more false negatives than false positives (i.e., it finds harder to claim the property is satisfied when it is actually not the case), which may be important for safety properties.

So, in summary, the field calculus monitor provides a reasonable approximation of an ideal monitor, but works in a decentralised, self-healing fashion, with devices providing an evaluation of the property by their local perspective, but still achieving (eventual) collective coherence through the continuous coordination with neighbours as regulated by the aggregate specification.

4.4. Discussion

The experiments demonstrate the technical validity of the field calculus solution for the monitoring of SLCS properties. While "eventual" stabilisation of the monitor outcome to the expected correct values is guaranteed by the theorems in previous section, this empirical evaluation actually shows that even in systems with quite dynamic topology, the inherent error in prediction remains to an acceptable bound. Indeed, Theorem 3 shows that a field calculus monitor obtained by translation of a SLCS formula stabilises to the truth value of that logical formula, *after a sufficient number of rounds with no changes*. However, in many practical application scenarios, small changes may happen almost continuously; in such circumstances, Theorem 3 does not help, since its premises may get invalidated very soon. The experimental evaluation shows that also in a scenario characterised by mobility and therefore continuous perturbations (as induced by changes in topology), the truth values computed by the monitor are sufficiently close to the ideal truth values at each instant (where the level of "sufficiency" depends on the particular application). In other words, the experimental results outlined in Section 4.3 show that the approach *can* cover practical cases that go beyond the hypothesis of formal theorems.

We stress that the monitor exercised in this section is *distributed* (actually, *decentralised*) and *self-adaptive*. Notice that distribution is simulated: we did not perform an actual large-scale deployment but rather simulated a network of smartphones and access points communicating based on spatial proximity (according to a typical Wi-Fi range). The logical computational model of the field calculus is intrinsically decentralised in control: it assumes each node repeatedly runs the program and shares coordination messages with neighbours—there is no centralised entity orchestrating the system. Adaptiveness is driven by the field calculus program and stems from the specifics of the execution model (as it may affect the relative order of operations), the environmental dynamics, and the evolution of the state of the system. In a computational round, a device executes the field calculus program against an up-to-date context that considers sensor data and recent messages received from neighbours. A change in the context will probably cause the device to also change its state, its outputs, and the messages it will send to neighbours—which in turn will adapt to their new context. In this way, local changes propagate through neighbourhoods to affect increasingly non-local portions of the network, ultimately affecting other localities. Essentially, the field calculus program is responsible for organising local adaptations such that they bring to the desired globally distributed state. In the case study, a portion of the program is responsible for computing and evaluating the level of density; as the people with the smartphones move in the city, the system topology changes, and such a density level (a distributed field, which has potentially different values for different nodes of the network) changes as well. The SLCS property under evaluation also adapts as the estimated density level and the topology of the system changes.

Concerning performance, most specifically, the field calculus monitor necessarily "follows" the oracle with some delay. The reactivity of the system can be regulated through proper parameterisation and algorithmic optimisations. For instance, components affecting how fast the system can respond to perturbations include, e.g., the frequency of firings and communications, the time for which neighbour data is considered valid, the estimation of the network diameter, and the particular gradient algorithm (Audrito et al., 2018b) adopted (which is used to set up the distributed structure for information propagation and collection). Of course, any application scenario is potentially different, and the above parameters should be tuned accordingly to the expected levels of variability. Additionally, there is a sort of algorithmic inertia that should be taken into account: for instance, non-reachability takes a number of *diameter* rounds to be proved; so, countermeasures could be taken to "delay" invalidation of results. Also, network partitioning may be particularly problematic: consider an *unsafe partition* whose connected set does not include a safe node. A node moving from such an unsafe partition to a safe partition while touching, at the same time, a non-dangerous, safe cluster and a dangerous cluster, can compromise the latter if stale data is not removed and is immediately used to contribute to a property evaluation decision (c.f. **`interior`**). In this case, delaying contributions and short retention windows for neighbour data could help to mitigate disruptions.

Regarding the deployment and operational execution of aggregate systems, various options are available (Viroli et al., 2016). Devices may locally run the field calculus program and directly communicate with neighbours or delegate these tasks to other (e.g., more powerful) devices—in that case, however, they must, at a minimum, provide sensor data and receive output/actuation data. So, in this latter view, access points may be useful to provide neighbourhood connectivity extending normal Wi-Fi range of smartphones as well as fog-level computing support.

4.5. Further considerations on applicability

In light of the above considerations, we can finally discuss the *suitability* of the approach for various systems and scenarios. First of all, due to the delays involved, the approach may not be adequate for applications where real-time exact responses are expected (although preliminary results on the applicability of aggregate computing techniques to real-time scenarios have already been investigated in Audrito et al. (2018c)). However, this limitation holds in general for decentralised solutions where global knowledge has to incrementally build up from local knowledge. Instead, the approach can be particularly useful when some delay or error can be tolerated. Typically, such a tolerance depends on application requirements and is related to constraints and situations to withstand. For instance, scenarios characterised by adversarial conditions such as frequent changes able to potentially affect global properties (i.e., like the one considered in this section) require certain levels of reactivity for the monitoring system to be usable and useful at all. In these kinds of systems, decentralised monitoring approaches should be evaluated on a case-by-case basis.

On the other hand, the proposed approach nicely fits scenarios characterised by moderate change and where approximated responses are acceptable during transient phases. Decentralised approaches are also favourable in very large-scale settings, to avoid single-points-of-failures, when there is no infrastructure in place, and as fallback solutions where centralised servers become unavailable (c.f. graceful degradation).

Additionally, recall that reactivity and precision of field calculus SLCS monitors are related to a few network characteristics. For instance, the larger the network diameter (i.e., number of hops of the longest shortest path), the longer it takes for information to reach the whole network, hence directly affecting the timing of **`somewhere`** and **`reaches`**. This problem could be mitigated, e.g., by applying the divide-et-impera principle and organising the system into bounded working areas (Casadei et al., 2019b), possibly overlapping as per Casadei et al. (2021), to enable multiple monitoring slices, and constructed by leveraging gradients originating from a selection of the safe places. The hop-by-hop propagation time is also affected by the frequency with which computational rounds are executed by the devices—which, in general, may depend on device energy levels, technical requirements and limitations (c.f. LoRaWAN systems (Adelantado et al., 2017)), or design choices. The relative frequency of round executions and environment dynamics also determines the reactivity with which inputs are con-

sidered. Last but not least, higher density (i.e., average number of neighbours) levels can provide higher stability, as changes are less likely to be disruptive.

5. Related work

As discussed in Section 2, our idea of specifying a property in a modal logic and then evaluating it step-wise is most closely related with the field of runtime verification (Leucker and Schallhart, 2009): while in runtime verification properties are usually specified in a temporal logic with operators such as *always* and *eventually*, here the modalities are spatial like or *everywhere* and *somewhere*. Nonetheless, the core aspect of runtime verification, the evaluation of properties as the system runs, is preserved in our setting: from the perspective of each device, the property is evaluated to a truth value every step, where a step here corresponds to a firing on each device.

While traditionally runtime verification considers evaluating a property on a single trace, the extension to distributed runtime verification makes the participation of multiple entities explicit.We discuss this sub-field in the following.

5.1. Distributed runtime verification and spatial logics

Distributed runtime verification lifts the concept of runtime verification to distributed systems (see (Francalanza et al., 2018) for a survey), finding applications in the following areas: *(i)* observing distributed computations & expressiveness (specifications over the distributed systems), *(ii)* analysis decomposition (coupled composition of system- and monitoring components), *(iii)* exploiting parallellism (in the evaluation of monitors), *(iv)* fault tolerance and *(v)* efficiency gains (by optimising communication). In the following, we discuss some works in this area are related to our aim, though none of them address the dynamisms at the same level; most assume a fixed number of participants and fixed communication topology.

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. (2004) introduce PT-DTL to specify distributed properties in a past time temporal logic, where 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. Both 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.

Our work is more closely related to those that have grown out of the spatial logics community, and moved into the area of runtime verification. In Nenzi et al. (2018), Signal Temporal Logic (STL) for real-valued signals takes inspiration from SLCS and is extended with the spatial modalities *somewhere* and *bounded surround* into Spatio-STL (SSTL). A monitoring algorithm is presented and its

implementation evaluated, though, in contrast to our work, the topology of the system is considered fixed. This is addressed by Bartocci et al. (2017) with the Spatio-Temporal Reach and Escape Logic (STREL), which in turn extends the above SSTL logic with two further modalities, *reach* and *escape*, which are designed as local properties, only taking into account neighbours. A monitoring algorithm is presented. To the best of our knowledge, no distributed algorithm has been presented yet to monitor distributed properties in large-scale dynamic networks.

5.2. Runtime verification of self-adaptive systems

Techniques for runtime verification have also been investigated in the context of self-adaptive systems, where the related problem of monitoring is particularly crucial to drive proper adaptation. In Borda et al. (2018), specifications expressed in a higher-order process language for adaptive CPSs are translated to FDR (Failures-Divergences Refinement) to refinement-check requirement satisfaction. Another approach, *Lotus@Runtime* (Barbosa et al., 2017) addresses verification of self-adaptive systems, modelled as (probabilistic) labelled transition systems, by checking reachability properties on execution traces—which must be generated, e.g., through instrumentation or aspect-oriented techniques. In Tahara et al. (2017), CAMPer is proposed, a property verifier for Component Aspect Models (CAM) UML profile that uses Maude for expressing behaviours and verifying dynamic evolution processes; however, unlike our approach, this is not applied to large-scale scenarios, and does not deal with decentralised control. Calinescu et al. (2017) provide a survey of quantitative model checking approaches for the (re-)verification of QoS properties after system, environment, or requirements change. Our field-based approach to coordination, in particular, naturally addresses the challenges of "continual re-assessment" which are stressed in the above work. Moreover, our approach captures properties to be verified as *executable specifications*, and the decentralised, self-healing monitor is directly "implied" from these, since their continuous, distributed interpretation yields the needed computation and communication activities for their local evaluation.

5.3. Ensembles of devices and aggregate computing

Several foundational calculi for describing interaction of devices in distributed systems have been proposed, mostly rooted on the archetype process algebra for mobility, the π-calculus (Milner et al., 1992a,b). Approaches like ambient calculus (Cardelli and Gordon, 2000), Bigraphs (Milner, 2006), 3π (Cardelli and Gardner, 2010), SCEL (De Nicola et al., 2013), and many others, provide mathematically concise foundations for capturing the interaction of groups in complex environments, featuring a shared-space abstraction by which multiple processes can interact in a decoupled way. However, they do not feature mechanisms for capturing the overall behaviour of an ensemble by abstracting over the single devices as with the field calculus, and for making such a behaviour compositional as required by the formulation of spatial properties. This makes them

quite low level for the purpose of expressing distribute monitors automatically generated from SLCS specifications.

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) and TinyLime (Curino et al., 2005) are information systems 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 be 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. Recent works have also adopted this field calculus as a *lingua franca* to investigate formal properties of resiliency to environment changes (Audrito et al., 2018c; Nishiwaki, 2016; Viroli et al., 2018), and to device distribution (Beal et al., 2017).

6. Conclusion and future work

In this paper we provided a natural translation of properties expressed in SLCS logic, a spatial logics with topological modal operators, into distributed programs for monitoring such properties. Such programs define a repetitive task to be executed by local monitors hosted in each device of the network, resulting in a coordinated behaviour that altogether computes local validity of the SLCS formula, and self-adapt optimally after changes in network topology or of truth values of the atomic propositions in the formula. This adaptation process is modelled through *self-stabilisation* and proved correct in Theorem 3. Additionally, local monitors run using local memory, message size and computation time that are all linear in the size of the formula (c.f. Theorem 2). Critical to achieve these results is the usage of aggregate computing (Beal et al., 2015) and the field calculus model (Audrito et al., 2019), which provided: *(i)* a functional programming model easily expressing the translation in a syntax-directed way, *(ii)* operators and libraries to easily capture the ability to monitor SLCS spatial operators, *(iii)* a programming language (Protelis, Pianini et al., 2015) and simulator (Alchemist, Pianini et al., 2013) to perform empirical evaluation in realistic scenarios, and finally *(iv)* a characterisation of self-stabilising field calculus programs as of Viroli et al. (2018), by which we could state resiliency of the runtime verification process. In particular, we also evaluated the approach in a large-scale crowd safety scenario, and showed that, even in environments characterised by nearly continuous and possibly disruptive change, essentially undermining the self-stabilisation requirements, the decentralised monitor still nicely approximates an oracle monitor.

Future works will be mainly devoted to capture more powerful monitoring processes, by considering more expressive spatial logics, as well as logics addressing spatio-temporal aspects. Additionally, we will seek for platform support for field calculus programs, encompassing the opportunistic usage of cloud as well as edge resources, along the line of (Viroli et al., 2016).

Acknowledgements

We thank the JSS anonymous reviewers for insightful comments and suggestions for improving the presentation.

References

Adelantado, F., Vilajosana, X., Tuset-Peiró, P., Martínez, B., Melià-Seguí, J., Watteyne, T., 2017. Understanding the limits of LoRaWAN. IEEE Commun. Mag. 55, 34–40.

Aiello, M., 2002. Spatial reasoning: theory and practice. Ph.D. thesis. Institute for Logic, Language and Computation, University of Amsterdam.

Anzengruber, B., Pianini, D., Nieminen, J., Ferscha, A., 2013. Predicting social density in mass events to prevent crowd disasters, in: International Conference on Social Informatics, Springer. pp. 206–215. doi:10.1007/978-3-319-03260-3_18.

Audrito, G., Beal, J., Damiani, F., Pianini, D., Viroli, M., 2020. Field-based coordination with the share operator. Log. Methods Comput. Sci. 16. doi: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 (COORDINATION), Springer. pp. 1–20. doi:10.1007/978-3-319-92408-3_1.

Audrito, G., Damiani, F., Viroli, M., 2018b. Optimal single-path information propagation in gradient-based algorithms. Sci. Comput. Program. 166, 146–166. doi:10.1016/j.scico.2018.06.002.

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. doi:10.1109/RTSS.2018.00013.

Audrito, G., Damiani, F., Viroli, M., Casadei, R., 2016. Run-time management of computation domains in field calculus, in: 1st Intl. Workshops on Foundations and Applications of Self* Systems (FAS*W), IEEE. pp. 192–197. doi:10.1109/FAS-W.2016.50.

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

Barbosa, D.M., Lima, R.G.D.M., Maia, P.H.M., Costa, E., 2017. Lotus@ runtime: a tool for runtime monitoring and verification of self-adaptive systems, in: 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE. pp. 24–30.

Bartocci, E., Bortolussi, L., Loreti, M., Nenzi, L., 2017. Monitoring mobile and spatially distributed cyber-physical systems, in: Talpin, J., Derler, P., Schneider, K. (Eds.), 15th ACM-IEEE Intl. Conf. on Formal Methods and Models for System Design, MEMOCODE 2017, pp. 146–155. doi:10.1145/3127041.3127050.

Bauer, A., Falcone, Y., 2016. Decentralised LTL monitoring. Formal Methods in System Design 48, 46–93. doi:10.1007/s10703-016-0253-8.

Bauer, A., Leucker, M., Schallhart, C., 2011. Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20, 14:1–14:64. doi:10.1145/2000799.2000800.

Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N., 2013. Organizing the aggregate: Languages for spatial computing, in: Mernik, M. (Ed.), Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global. chapter 16, pp. 436–501. doi:10.4018/978-1-4666-2092-6.ch016. Longer version available at: http://arxiv.org/abs/1202.5509.

Beal, J., Pianini, D., Viroli, M., 2015. Aggregate programming for the Internet of Things. IEEE Computer 48, 22–30. doi:10.1109/MC.2015.261.

Beal, J., Viroli, M., 2014. Building blocks for aggregate programming of self-organising applications, in: 8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW), IEEE Computer Society. pp. 8–13. doi:10.1109/SASOW.2014.6.

Beal, J., Viroli, M., Pianini, D., Damiani, F., 2017. Self-adaptation to device distribution in the Internet of Things. ACM Transaction on Autonomous and Adaptive Systems 12, 12:1–12:29. doi:10.1145/3105758.

Ben-Ari, M., Pnueli, A., Manna, Z., 1983. The temporal logic of branching time. Acta Informatica 20, 207–226. doi:10.1007/BF01257083.

Bennaceur, A., Ghezzi, C., Tei, K., Kehrer, T., Weyns, D., Calinescu, R., Dustdar, S., Hu, Z., Honiden, S., Ishikawa, F., Jin, Z., Kramer, J., Litoiu, M., Loreti, M., Moreno, G., Müller, H., Nenzi, L., Nuseibeh, B., Pasquale, L., Reisig, W., Schmidt, H., Tsigkanos, C., Zhao, H., 2019. Modelling and

analysing resilient cyber-physical systems, in: 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 70–76. doi:10.1109/SEAMS.2019.00018.

van Benthem, J., Bezhanishvili, G., 2007. Modal logics of space, in: Aiello, M., Pratt-Hartmann, I., van Benthem, J. (Eds.), Handbook of Spatial Logics. Springer, pp. 217–298. doi:10.1007/978-1-4020-5587-4_5.

Borda, A., Pasquale, L., Koutavas, V., Nuseibeh, B., 2018. Compositional verification of self-adaptive cyber-physical systems, in: 2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE. pp. 1–11.

Calinescu, R., Gerasimou, S., Johnson, K., Paterson, C., 2017. Using runtime quantitative verification to provide assurance evidence for self-adaptive software, in: Software Engineering for Self-Adaptive Systems III. Assurances. Springer, pp. 223–248.

Cardelli, L., Gardner, P., 2010. Processes in space, in: 6th Conference on Computability in Europe, Springer. pp. 78–87. doi:10.1007/978-3-642-13962-8.

Cardelli, L., Gordon, A.D., 2000. Mobile ambients. Theoretical Computer Science 240, 177 – 213. doi:10.1016/S0304-3975(99)00231-5.

Casadei, R., Fortino, G., Pianini, D., Russo, W., Savaglio, C., Viroli, M., 2019a. A development approach for collective opportunistic edge-of-things services. Information Sciences 498, 154–169.

Casadei, R., Pianini, D., Viroli, M., Natali, A., 2019b. Self-organising coordination regions: A pattern for edge computing, in: COORDINATION, Springer. pp. 182–199.

Casadei, R., Viroli, M., 2016. Towards aggregate programming in Scala, in: First Workshop on Programming Models and Languages for Distributed Computing, ACM, New York, NY, USA. pp. 5:1–5:7. doi:10.1145/2957319.2957372.

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

Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loreti, M., Massink, M., 2018. Spatio-temporal model checking of vehicular movement in public transport systems. STTT 20, 289–311. doi:10.1007/s10009-018-0483-8.

Ciancia, V., Latella, D., Loreti, M., Massink, M., 2014. Specifying and verifying properties of space, in: Díaz, J., Lanese, I., Sangiorgi, D. (Eds.), 8th IFIP International Conference in Theoretical Computer Science (TCS), Springer. pp. 222–235. doi:10.1007/978-3-662-44602-7_18.

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

Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P., 2005. Mobile data collection in sensor networks: The TinyLime middleware. Elsevier Pervasive and Mobile Computing Journal 4, 446–469. doi:10.1016/j.pmcj.2005.08.003.

De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R., 2013. A language-based approach to autonomic computing, in: Formal Methods for Components and Objects. FMCO 2011, Springer. pp. 25–48. doi:10.1007/978-3-642-35887-6_2.

Emerson, E.A., 1990. Temporal and modal logic, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics. Elsevier and MIT Press, pp. 995–1072. doi:10.1016/b978-0-444-88074-1.50021-4.

Francalanza, A., Pérez, J.A., Sánchez, C., 2018. Runtime verification for decentralised and distributed systems, in: Bartocci, E., Falcone, Y. (Eds.), Lectures on Runtime Verification - Introductory and Advanced Topics. Springer. volume 10457 of *Lecture Notes in Computer Science*, pp. 176–210. doi: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. Springer. volume 3566 of *Lecture Notes in Computer Science*, pp. 137–152. doi:10.1007/11527800_11.

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

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

Lluch-Lafuente, A., Loreti, M., Montanari, U., 2017. Asynchronous distributed execution of fixpoint-based computational fields. Logical Methods in Computer Science 13. doi:10.23638/LMCS-13(1:13)2017.

Mamei, M., Zambonelli, F., 2009. Programming pervasive and mobile computing applications: The TOTA approach. ACM Trans. on Software Engineering Methodologies 18, 1–56. doi:10.1145/1538942.1538945.

Michail, D., Kinable, J., Naveh, B., Sichi, J.V., 2020. JGraphT - A Java library for graph data structures and algorithms. ACM Trans. Math. Softw. 46, 16:1–16:29. doi:10.1145/3381449.

Milner, R., 2006. Pure bigraphs: Structure and dynamics. Information and Computation 204, 60 – 122. doi:10.1016/j.ic.2005.07.003.

Milner, R., Parrow, J., Walker, D., 1992a. A calculus of mobile processes, i. Information and Computation 100, 1 – 40. doi:10.1016/0890-5401(92)90008-4.

Milner, R., Parrow, J., Walker, D., 1992b. A calculus of mobile processes, ii. Information and Computation 100, 41 – 77. doi:10.1016/0890-5401(92)90009-5.

Mo, Y., Audrito, G., Dasgupta, S., Beal, J., 2020. A resilient leader election algorithm via aggregate computing blocks, in: Proceedings of the IFAC World Congress. To appear.

Nenzi, L., Bortolussi, L., Ciancia, V., Loreti, M., Massink, M., 2018. Qualitative and quantitative monitoring of spatio-temporal properties with SSTL. Logical Methods in Computer Science 14. doi:10.23638/LMCS-14(4:2)2018.

Newton, R., Welsh, M., 2004. Region streams: Functional macroprogramming for sensor networks, in: Workshop on Data Management for Sensor Networks, pp. 78–87. doi: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. doi:10.1109/FAS-W.2016.51.

Pianini, D., Beal, J., Viroli, M., 2016. Improving gossip dynamics through overlapping replicates, in: Lluch-Lafuente, A., Proença, J. (Eds.), 18th International Conference on Coordination Models and Languages (COORDINATION), Springer. pp. 192–207. doi:10.1007/978-3-319-39519-7_12.

Pianini, D., Montagna, S., Viroli, M., 2013. Chemical-oriented simulation of computational systems with ALCHEMIST. J. Simulation 7, 202–215. doi:10.1057/jos.2012.27.

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. doi:10.1145/2695664.2695913.

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. doi:10.1109/ICSE.2004.1317464.

Tahara, Y., Ohsuga, A., Honiden, S., 2017. Formal verification of dynamic evolution processes of uml models using aspects, in: Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press. pp. 152–162.

Viroli, M., Audrito, G., Beal, J., Damiani, F., Pianini, D., 2018. Engineering resilient collective adaptive systems by self-stabilisation. ACM Transactions on Modelling and Computer Simulation 28, 16:1–16:28. doi:10.1145/3177774.

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

Viroli, M., Casadei, R., Pianini, D., 2016. On execution platforms for large-scale aggregate computing, in: Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct, ACM. pp. 1321–1326.

Whitehouse, K., Sharp, C., Culler, D.E., Brewer, E.A., 2004. Hood: A neighborhood abstraction for sensor networks, in: 2nd International Conference on Mobile Systems, Applications, and Services, ACM / USENIX. pp. 99–110. doi:10.1145/990064.990079.

Winskel, G., 1982. Event structure semantics for ccs and related languages, in: Nielsen, M., Schmidt, E.M. (Eds.), Automata, Languages and Programming, Springer. pp. 561–576. doi:10.1007/BFb0012800.

Appendix A. Operational semantics of the field calculus

To simplify the notation, in the presentation we assume a fixed program P. We say that "device δ fires" to mean that the main expression $\mathtt{e}_{\mathtt{main}}$ of P is evaluated on δ at a particular firing. The computation that takes place on a single device is formalised by a big-step semantics (given in Appendix A.1), while the overall network computation is formalised by a small-step semantics (given in Appendix A.2).

Appendix A.1. Device semantics

The result of a device firing is an ordered tree of values θ, called a *value-tree*. It tracks the results of all evaluated subexpressions of $\mathtt{e}_{\mathtt{main}}$. Such a value-tree is made available to δ's neighbours for their subsequent firing (including δ itself, so as to support a form of state across firings). Each device collects the recently-received value-trees of neighbours into a map Θ from device identifiers to value-trees (written $\overline{\delta} \mapsto \overline{\theta}$ as short for $\delta_1 \mapsto \theta_1, \ldots, \delta_n \mapsto \theta_n$), called a *value-tree environment*. The outcome of the evaluation will depend on those value-trees. The syntax of value-trees and value-tree environments is given in Figure A.12 (top).

Example 6. The graphical representation of the value trees $1\langle 3\langle\rangle, 2\langle\rangle\rangle$ and $1\langle 3\langle\rangle, 2\langle 7\langle\rangle, 2\langle\rangle, 5\langle\rangle\rangle\rangle$ is as follows:

For sake of readability, we sometimes write the value $\mathtt{v}$ as short for the value-tree $\mathtt{v}\langle\rangle$. Following this convention, the value-tree $1\langle 3\langle\rangle, 2\langle\rangle\rangle$ is shortened to $1\langle 3, 2\rangle$, and the value-tree $1\langle 3\langle\rangle, 2\langle 7\langle\rangle, 2\langle\rangle, 5\langle\rangle\rangle\rangle$ is shortened to $1\langle 3, 2\langle 7, 2, 5\rangle\rangle$.

The judgement that describes the firing of a device, defined in Figure A.12 (bottom), is $\delta; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta$, where: *(i)* δ is the identifier of the device that fires; *(ii)* Θ is the environment collecting the value-trees produced by the most recent evaluation of (an expression corresponding to) $\mathtt{e}$ on δ's neighbours; *(iii)* $\mathtt{e}$ is a closed run-time expression (i.e., a closed expression that may contain neighbouring values); *(iv)* θ is the value-tree representing the values computed for all the expressions encountered during the evaluation of $\mathtt{e}$—the root of the value tree θ is the value computed for expression $\mathtt{e}$. It is denoted by $\rho(\theta)$, where ρ is the auxiliary function defined in Figure A.12 (second frame).

The operational semantics rules resemble standard rules for functional languages, however they are extended to ensure that each subexpression $\mathtt{e}'$ of $\mathtt{e}$ is evaluated with respect to the value-tree environment Θ' obtained from Θ by extracting (when present) the corresponding subtree in the value-trees in the range of Θ. This process, called *alignment*, is modelled by the auxiliary function π defined in Figure A.12 (second frame). This function has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i-th subtree of θ; while $\pi^{\ell}(\theta)$ extracts the last subtree of θ, *if* the root of the first subtree of θ is equal to the local (Boolean) value ℓ (thus implementing a filter specifically designed for the `if` construct). Auxiliary functions ρ and π apply pointwise on value-tree environments, as defined in Figure A.12 (second frame).

Rules [E-LOC] and [E-FLD] model the evaluation of expressions that are either a local value or a neighbouring value, respectively. In particular, rule [E-FLD] restricts the domain of a neighbouring value to the only set of neighbour devices as reported in Θ.

Rule [E-B-APP] models the application of built-in functions (including measurement variables and interactions with the external world via sensors and actuators), that is, of expressions of the form $\mathtt{b}(\mathtt{e}_1 \cdots \mathtt{e}_n)$, where $n \geq 0$. The produced value-tree is $\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle$, where $\theta_1, \ldots, \theta_n$ are the value-trees produced by the evaluation of the actual parameters $\mathtt{e}_1, \ldots, \mathtt{e}_n$ and $\mathtt{v}$ is the value returned by the function. The rule exploits the special auxiliary function $(\!|\mathtt{b}|\!)^{\delta,\Theta}_{\sigma}$ (whose actual definition is abstracted away) which ensures that $(\!|\mathtt{b}|\!)^{\delta,\Theta}_{\sigma}(\overline{\mathtt{v}})$ computes the result of applying built-in function $\mathtt{b}$ to values $\overline{\mathtt{v}}$ in the current environment of the device δ. The built-in 0-ary function `self` gets evaluated to the current device identifier (i.e., $(\!|\mathtt{self}|\!)^{\delta,\Theta}_{\sigma}() = \delta$), and mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $(\!|-|\!)^{\delta,\Theta}_{\sigma}(3, 2) = 1$).

Example 7. Evaluating the expression $-(\mathtt{3}, \mathtt{2})$ produces the value-tree $1\langle 3, 2\rangle$. The value of the whole expression, 1, has been computed by using rule [E-B-APP] to evaluate the application of the multiplication operator $-$ to the values 3 (the root of the first subtree of the value-tree) and 2 (the root of the second subtree of the value-tree).

Rule [E-D-APP] models the application of a user-defined function, that is, of expressions of the form $\mathtt{d}(\mathtt{e}_1 \ldots \mathtt{e}_n)$, where $n \geq 0$. It resembles rule [E-B-

Value-trees and value-tree environments:

$\theta \quad ::= \quad \mathtt{v}\langle\overline{\theta}\rangle$ value-tree

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

Auxiliary functions:

$\rho(\mathtt{v}\langle\overline{\theta}\rangle) = \mathtt{v}$

$\pi_i(\mathtt{v}\langle\theta_1, \ldots, \theta_n\rangle) = \theta_i \quad \text{if } 1 \leq i \leq n$ $\qquad \pi^\ell(\mathtt{v}\langle\theta_1, \theta_2\rangle) = \theta_2 \quad \text{if } \rho(\theta_1) = \ell$

$\pi_i(\theta) = \bullet \quad \text{otherwise}$ $\qquad \pi^\ell(\theta) = \bullet \quad \text{otherwise}$

$$\text{For } aux \in \rho, \pi_i, \pi^\ell : \begin{cases} aux(\delta \mapsto \theta) & = & \delta \mapsto aux(\theta) & \text{if } aux(\theta) \neq \bullet \\ aux(\delta \mapsto \theta) & = & \bullet & \text{if } aux(\theta) = \bullet \\ aux(\Theta, \Theta') & = & aux(\Theta), aux(\Theta') & \end{cases}$$

$args(\mathtt{d}) = \overline{\mathtt{x}} \quad \text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$ $\qquad body(\mathtt{d}) = \mathtt{e} \quad \text{if } \mathtt{def\ d}(\overline{\mathtt{x}})\ \{\mathtt{e}\}$

$$\phi_0[\phi_1] = \phi_2 \text{ where } \phi_2(\delta) = \begin{cases} \phi_1(\delta) & \text{if } \delta \in \mathbf{dom}(\phi_1) \\ \phi_0(\delta) & \text{otherwise} \end{cases}$$

Syntactic shorthands:

$\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta}$ where $|\overline{\mathtt{e}}| = n$ for $\begin{array}{l} \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \\ \delta; \pi_n(\Theta); \sigma \vdash \mathtt{e}_n \Downarrow \theta_n \end{array}$

$\rho(\overline{\theta})$ where $|\overline{\theta}| = n$ for $\rho(\theta_1), \ldots, \rho(\theta_n)$

$\overline{\mathtt{x}} := \rho(\overline{\theta})$ where $|\overline{\mathtt{x}}| = n$ for $\mathtt{x}_1 := \rho(\theta_1) \ \ldots \ \mathtt{x}_n := \rho(\theta_n)$

Rules for expression evaluation: $\boxed{\delta; \Theta; \sigma \vdash \mathtt{e} \Downarrow \theta}$

$$\frac{}{\delta; \Theta; \sigma \vdash \ell \Downarrow \ell\langle\rangle}\ [\text{E-LOC}] \qquad \frac{\phi' = \phi|_{\mathbf{dom}(\Theta) \cup \{\delta\}}}{\delta; \Theta; \sigma \vdash \phi \Downarrow \phi'\langle\rangle}\ [\text{E-FLD}]$$

$$\frac{\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \qquad \mathtt{v} = (\!|\mathtt{b}|\!)_\sigma^{\delta,\Theta}(\rho(\overline{\theta}))}{\delta; \Theta; \sigma \vdash \mathtt{b}(\overline{\mathtt{e}}) \Downarrow \mathtt{v}\langle\overline{\theta}\rangle}\ [\text{E-B-APP}]$$

$$\frac{\delta; \overline{\pi}(\Theta); \sigma \vdash \overline{\mathtt{e}} \Downarrow \overline{\theta} \qquad \delta; \Theta; \sigma \vdash body(\mathtt{d})[args(\mathtt{d}) := \rho(\overline{\theta})] \Downarrow \theta'}{\delta; \Theta; \sigma \vdash \mathtt{d}(\overline{\mathtt{e}}) \Downarrow \rho(\theta')\langle\overline{\theta}, \theta'\rangle}\ [\text{E-D-APP}]$$

$$\frac{\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta \qquad \phi = \rho(\pi_1(\Theta))[\delta \mapsto \rho(\theta)]}{\delta; \Theta; \sigma \vdash \mathtt{nbr}\{\mathtt{e}\} \Downarrow \phi\langle\theta\rangle}\ [\text{E-NBR}]$$

$$\frac{\begin{array}{l} \delta; \pi_1(\Theta); \sigma \vdash \mathtt{e}_1 \Downarrow \theta_1 \qquad \phi' = \rho(\pi_2(\Theta)) \qquad \phi = (\delta \mapsto \rho(\theta_1))[\phi'] \\ \delta; \pi_2(\Theta); \sigma \vdash \mathtt{e}_2[\mathtt{x} := \phi] \Downarrow \theta_2 \end{array}}{\delta; \Theta; \sigma \vdash \mathtt{share}(\mathtt{e}_1)\{(\mathtt{x}) \Rightarrow \mathtt{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1, \theta_2\rangle}\ [\text{E-SHARE}]$$

$$\frac{\delta; \pi_1(\Theta); \sigma \vdash \mathtt{e} \Downarrow \theta_1 \quad \rho(\theta_1) \in \{\mathtt{true}, \mathtt{false}\} \quad \delta; \pi^{\rho(\theta_1)}(\Theta); \sigma \vdash \mathtt{e}_{\rho(\theta_1)} \Downarrow \theta}{\delta; \Theta; \sigma \vdash \mathtt{if(e)\{e_{true}\}\ else\ \{e_{false}\}} \Downarrow \rho(\theta)\langle\theta_1, \theta\rangle}\ [\text{E-IF}]$$

Figure A.12: Big-step operational semantics for expression evaluation.

APP] while producing a value-tree with one more subtree θ', which is produced by evaluating the body of the function d (denoted by *body*(d)) substituting the formal parameters of the function (denoted by *args*(d)) with the values obtained

evaluating $\mathtt{e}_1, \ldots \mathtt{e}_n$.

Rule [E-NBR] first collects neighbours' values for expressions $\mathtt{e}$ as $\phi = \rho(\pi_1(\Theta))$, then evaluates $\mathtt{e}$ in δ and updates the corresponding entry in ϕ to produce its overall value.

Rule [E-SHARE] uses the notation $\phi_0[\phi_1]$, defined in Figure A.12 (second frame), to express "neighbouring value update": the updated neighbouring value $\phi_2 = \phi_0[\phi_1]$ has $\mathbf{dom}(\phi_2) = \mathbf{dom}(\phi_0) \cup \mathbf{dom}(\phi_1)$ and coincides with ϕ_1 on its domain, or with ϕ_0 otherwise. The evaluation rule [E-SHARE] produces a value-tree with two branches (for $\mathtt{e}_1$ and $\mathtt{e}_2$ respectively). First, it evaluates $\mathtt{e}_1$ with respect to the corresponding branches of neighbours $\pi_1(\Theta)$ obtaining θ_1. Then, it collects the results for the construct from neighbours into the neighbouring value $\phi' = \rho(\pi_2(\Theta))$. In case ϕ' does not have an entry for δ, $\rho(\theta_1)$ is used obtaining $\phi = (\delta \mapsto \rho(\theta_1))[\phi']$. Finally, ϕ is substituted for $\mathtt{x}$ in the evaluation of $\mathtt{e}_2$ (with respect to the corresponding branches of neighbours $\pi_2(\Theta)$) obtaining θ_2, setting $\rho(\theta_2)$ to be the overall value.

Example 8. Consider a program consisting of the body of function `gossipEver` (introduced Example 1) where the occurrence of the parameter `alarm` has been replaced by the call to a built-in `sense` that returns the value of a Boolean sensor:

```
share (false) { (old) => anyHoodPlusSelf(old) || sense() }
```

Suppose that the program runs on a network of two mutually interconnect devices δ_0 and δ_1, and that device δ_0 first executes a firing with an empty environment Θ and with `sense()` returning `false`. The evaluation of the `share` construct proceeds by evaluating `false` into $\theta_1 = \mathtt{false}\langle\rangle$, gathering neighbour values into $\phi' = \bullet$ (no values are present), and adding the value for the current device obtaining $\phi = (\delta_0 \mapsto \mathtt{false})[\bullet] = \delta_0 \mapsto \mathtt{false}$. Finally, the evaluation completes with the result of $\mathtt{anyHoodPlusSelf}(\delta_0 \mapsto \mathtt{false}) \,||\, \mathtt{false}$ (which is $\mathtt{false}\langle\mathtt{false}\langle\delta_0 \mapsto \mathtt{false}\rangle, \mathtt{false}\rangle$) corresponding to θ_2 in rule [E-SHARE]. At the end of the firing, device δ_0 sends a broadcast message containing the result of its overall evaluation, and thus including $\theta^0 = \mathtt{false}\langle\mathtt{false}, \theta_2\rangle$.

Suppose now that device δ_1 receives the broadcast message and then executes a firing with $\Theta = (\delta_0 \mapsto \theta^0)$ and `sense()` returning `true`. The evaluation of the `share` constructs starts similarly as before with $\theta_1 = \mathtt{false}\langle\rangle$, $\phi' = \delta_0 \mapsto \mathtt{false}$, $\phi = \delta_0 \mapsto \mathtt{false}, \delta_1 \mapsto \mathtt{false}$. Then the body of the `share` is evaluated as $\mathtt{anyHoodPlusSelf}(\delta_0 \mapsto \mathtt{false}, \delta_1 \mapsto \mathtt{false}) \,||\, \mathtt{true}$ into $\theta_2 = \mathtt{true}\langle\mathtt{false}\langle\delta_0 \mapsto \mathtt{false}, \delta_1 \mapsto \mathtt{false}\rangle, \mathtt{true}\rangle$. At the end of the firing, device δ_1 broadcasts the result of its overall evaluation, including $\theta^1 = \mathtt{true}\langle\mathtt{false}, \mathtt{true}\langle\theta_2\rangle\rangle$.

Then, suppose that device δ_0 receives the broadcast from device δ_1 and then performs another firing with $\Theta = (\delta_0 \mapsto \theta^0, \delta_1 \mapsto \theta^1)$ and `sense()` returning `false`. As before, $\theta_1 = \mathtt{false}\langle\rangle$, $\phi = \phi' = \delta_0 \mapsto \mathtt{false}, \delta_1 \mapsto \mathtt{true}$ and the body is evaluated as $\mathtt{anyHoodPlusSelf}(\delta_0 \mapsto \mathtt{false}, \delta_1 \mapsto \mathtt{true}) \,||\, \mathtt{false}$ into $\theta_2 = \mathtt{true}\langle\mathtt{false}\langle\delta_0 \mapsto \mathtt{false}, \delta_1 \mapsto \mathtt{true}\rangle, \mathtt{false}\rangle$. Then device δ_0 broadcasts the overall result $\theta^2 = \mathtt{true}\langle\mathtt{false}, \mathtt{true}\langle\theta\rangle\rangle$.

System configurations and action labels:

Ψ	::=	$\overline{\delta} \mapsto \overline{\Theta}$	status field
Env	::=	$\langle \rightarrowtail, \Sigma \rangle$	environment
N	::=	$\langle Env; \Psi \rangle$	network configuration
act	::=	$\delta \mid env$	action label

Environment well-formedness:

$WFE(\langle \rightarrowtail, \Sigma \rangle)$ holds iff $\rightarrowtail \subseteq D \times D$ where $D = \mathbf{dom}(\Sigma)$

Transition rules for network evolution: $\boxed{N \xrightarrow{act} N}$

$$\text{[N-FIR]}\quad \frac{Env = \langle \rightarrowtail, \Sigma \rangle \qquad \overline{\delta} = \{\delta' \mid \delta \rightarrowtail \delta'\} \qquad \delta; F(\Psi)(\delta); \Sigma(\delta) \vdash \mathtt{e_{main}} \Downarrow \theta \qquad \Psi_1 = \overline{\delta} \mapsto \{\delta \mapsto \theta\}}{\langle Env; \Psi \rangle \xrightarrow{\delta} \langle Env; F(\Psi)[\Psi_1] \rangle}$$

$$\text{[N-ENV]}\quad \frac{WFE(Env') \qquad Env' = \langle \rightarrowtail, \overline{\delta} \mapsto \overline{\sigma} \rangle \qquad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle}$$

Figure A.13: Small-step operational semantics for network evolution.

Finally, suppose that (because op a change of network topology that took place before the last firing of device δ_0) device δ_1 does not receive that broadcast and filters out δ_0 from its list of neighbour before performing another firing with `sense()` returning `false` (which is different from the value returned while performing the previous fire of device δ_1) . Then, $\theta_1 = \mathtt{false}\langle\rangle$, $\phi' = \delta_1 \mapsto \mathtt{true}$, $\phi = (\delta_1 \mapsto \mathtt{false})[\delta_1 \mapsto \mathtt{true}] = \delta_1 \mapsto \mathtt{true}$, and the body is evaluated as $\mathtt{anyHoodPlusSelf}(\delta_1 \mapsto \mathtt{true})\,||\,\mathtt{false}$ which produces $\theta_2 = \mathtt{true}\langle\mathtt{true}\langle\delta_1 \mapsto \mathtt{true}\rangle, \mathtt{false}\rangle$ and leads to the overall result $\theta^3 = \mathtt{true}\langle\mathtt{false}, \mathtt{true}\langle\theta_2\rangle\rangle$.

Rule [E-IF] is almost standard, except that it performs domain restriction $\pi^{\mathtt{true}}(\Theta)$ (resp. $\pi^{\mathtt{false}}(\Theta)$) in order to guarantee that subexpression $\mathtt{e_{true}}$ is not matched against value-trees obtained from $\mathtt{e_{false}}$ (and vice-versa).

Appendix A.2. Network semantics

The overall network evolution is formalised by the small-step operational semantics given in Figure A.13 as a transition system on network configurations N. Figure A.13 (top) defines key syntactic elements to this end. Ψ models the overall status of the devices in the network at a given time, as a map from device identifiers to value-tree environments. $\rightarrowtail$ models *network topology* (a directed neighbouring graph), as in Definition 5. Σ models *sensor (distributed) state*, as a computational field $\overline{\delta} \mapsto \overline{\sigma}$ (c.f. Definition 6) mapping device identifiers to (local) sensors (i.e., sensor name/value maps denoted as σ). Then, *Env* (a couple of topology and sensor state) models the system's environment. Finally, a whole network configuration N is a couple of a status field and environment.

We use the following notation for status fields. Let $\overline{\delta} \mapsto \Theta$ denote a map from device identifiers $\overline{\delta}$ to the same value-tree environment Θ. Let $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\mathbf{dom}(\Theta_0) \cup \mathbf{dom}(\Theta_1)$ coinciding with

Θ_1 in the domain of Θ_1 and with Θ_0 otherwise. Let $\Psi_0[\Psi_1]$ denote the status field with the *same domain* as Ψ_0 made of $\delta \mapsto \Psi_0(\delta)[\Psi_1(\delta)]$ for all δ in the domain of Ψ_1, $\delta \mapsto \Psi_0(\delta)$ otherwise.

Thee are transitions $N \xrightarrow{act} N'$ of two kinds: firings, where *act* is the corresponding device identifier, and environment changes, where *act* is the special label *env*. This is formalised in Figure A.13 (bottom). Rule [N-FIR] models a firing at device δ: it takes the local value-tree environment filtered out of old values $F(\Psi)(\delta)$;[12] then by the single device semantics it obtains the device's value-tree θ,[13] which is used to update the system configuration of δ and of δ's neighbours.

Rule [N-ENV] takes into account the change of the environment to a new *well-formed* environment Env'—environment well-formedness is specified by the predicate $WFE(Env)$ in Figure A.13 (middle). Let $\overline{\delta}$ be the domain of Env'. First, a status field Ψ_0 is constructed by associating to all the devices of Env' the empty context $\emptyset$. Then, the existing status field Ψ is adapted to the new set of devices: $\Psi_0[\Psi]$ automatically handles removal of devices, map of new devices to the empty context, and retention of existing contexts in the other devices.

Example 9. Consider a network of devices running the program

```
share (false) { (old) => anyHoodPlusSelf(old) || sense() }
```

as introduced in Example 8. The network configuration illustrated at the beginning of Example 8 can be generated by applying rule [N-ENV] to the empty network configuration. I.e., we have $\langle\langle\emptyset,\emptyset\rangle;\emptyset\rangle \xrightarrow{env} \langle Env_0;\Psi_0\rangle$ where

$$Env_0 = \langle \rightarrowtail_0, \Sigma_0 \rangle,$$

$$\rightarrowtail_0 = (\delta_0 \mapsto \delta_1, \delta_1 \mapsto \delta_0),$$

$$\Sigma_0 = (\delta_0 \mapsto (\texttt{sense} \mapsto \texttt{false}), \delta_1 \mapsto (\texttt{sense} \mapsto \texttt{true})), \text{ and}$$

$$\Psi_0 = (\delta_0 \mapsto \emptyset, \delta_1 \mapsto \emptyset).$$

Then, the four firings of devices δ_0, δ_1, δ_0, δ_1 and the change of communication topology and sensor value (that took place between the second and the third firing) illustrated in Example 8 are modelled by the following transitions.

1. $\langle Env_0;\Psi_0\rangle \xrightarrow{\delta_0} \langle Env_0;\Psi'\rangle$, where
$$\Psi' = (\delta_0 \mapsto (\delta_0 \mapsto \theta^0),\ \delta_1 \mapsto (\delta_0 \mapsto \theta^0)).$$
2. $\langle Env_0;\Psi'\rangle \xrightarrow{\delta_1} \langle Env_0;\Psi''\rangle$, where
$$\Psi'' = (\delta_0 \mapsto (\delta_0 \mapsto \theta^0, \delta_1 \mapsto \theta^1),\ \delta_1 \mapsto (\delta_0 \mapsto \theta^0, \delta_1 \mapsto \theta^1)).$$

[12] Function $F(\Psi)$ in rule [N-FIR] models a filtering operation that clears out old stored values from the value-tree environments in Ψ, implicitly based on space/time tags.

[13] Termination of a device firing is clearly not decidable. However, without loss of generality for the results of this paper, we assume that any device firing is guaranteed to terminate in any environmental condition.

3. $\langle Env_0; \Psi'' \rangle \xrightarrow{env} \langle Env_1; \Psi'' \rangle$, where

$$Env_1 = \langle \emptyset, \Sigma_1 \rangle,$$
$$\Sigma_1 = (\delta_0 \mapsto (\texttt{sense} \mapsto \texttt{false}), \delta_1 \mapsto (\texttt{sense} \mapsto \texttt{false})).$$

4. $\langle Env_1; \Psi'' \rangle \xrightarrow{\delta_0} \langle Env_1; \Psi''' \rangle$, where

$$\Psi''' = (\delta_0 \mapsto (\delta_0 \mapsto \theta^2, \delta_1 \mapsto \theta^1),\ \delta_1 \mapsto (\delta_0 \mapsto \theta^0, \delta^1 \mapsto \theta^1)).$$

5. $\langle Env_1; \Psi''' \rangle \xrightarrow{\delta_1} \langle Env_1; \Psi'''' \rangle$, where

$$\Psi'''' = (\delta_0 \mapsto (\delta_0 \mapsto \theta^2, \delta_1 \mapsto \theta^1),\ \delta_1 \mapsto (\delta^1 \mapsto \theta^3)).$$

Appendix B. Proofs

Appendix B.1. Proof of Theorem 1

In this section, we prove that the operational semantics in Appendix A mirrors the message passing details of any LUIC augmented event structure (c.f. Definition 2). Namely, every system evolution $\mathcal{S}$ induces a Space-Time Value $\Phi = \langle \mathbf{E}, f \rangle$ (c.f. Definition 3). Therefore, $\mathcal{S}$ induces a LUIC augmented event structure $\mathbf{E}$ (c.f. Definition 2) describing its message passing details, as per the following definition.

Definition 13 (Space-Time Value Induced by a System Evolution)**.** Let $\mathcal{S} = N_0 \xrightarrow{act_1} \ldots \xrightarrow{act_n} N_n$ with $N_0 = \langle \emptyset, \emptyset; \emptyset \rangle$ be any system evolution. We say that:

- $D = \{\delta \mid \exists i.\ act_i = \delta\}$ are the device identifiers appearing in $\mathcal{S}$;
- $F^\delta = \langle i \leq n \mid act_i = \delta \rangle$ are the indexes of transitions applying rule [N-FIR];
- $E = \left\{ \langle \delta, i \rangle \mid \delta \in D \ \wedge\ 1 \leq i \leq \left| F^\delta \right| \right\}$ is the set of events in $\mathcal{S}$;
- $d : E \to D$ maps each event $\epsilon = \langle \delta, i \rangle$ to the device δ where it is happening;
- $\epsilon_1 \rightsquigarrow \epsilon_2$ where $\epsilon_k = \langle \delta_k, i_k \rangle$ and $j_1 = F^{\delta_1}_{i_1}$, $j_2 = F^{\delta_2}_{i_2}$ if and only if:
  - N_{j_1} has topology $\rightarrowtail$ such that $\delta_1 \rightarrowtail \delta_2$ (the message from ϵ_1 reaches δ_2),
  - there is no $j' \in (j_1; j_2)$ with $j' \in F^{\delta_1}$ and $N_{j'}$ with topology $\rightarrowtail$ such that $\delta_1 \rightarrowtail \delta_2$ (there are no more recent messages from δ_1 to ϵ_2),
  - for every $j' \in (j_1; j_2]$ with $j' \in F^{\delta_2}$ and $N_{j'}$ with status field Ψ, then $\delta_1 \in \mathbf{dom}(\Psi(\delta_2))$ (the message from ϵ_1 to δ_2 is not filtered out as obsolete before ϵ_2);
- $<$ is the transitive closure of $\rightsquigarrow$;
- $f : E \to \mathbf{V}$ is such that $f(\langle \delta, i \rangle) = \rho(\Psi(\delta)(\delta))$ where $N_{F^\delta_i} = \langle Env; \Psi \rangle$.

Then we say that the system evolution $\mathcal{S}$ *induces* the space-time value $\Phi = \langle \mathbf{E}, f \rangle$, where $\mathbf{E}$ is the LUIC augmented event structure $\langle E, \rightsquigarrow, <, d \rangle$.

Notice that the $\mathbf{E}$ and Φ defined above are unique given $\mathcal{S}$. Furthermore, as stated by the following theorem, the operational semantics is sufficiently expressive to model every possible message interaction describable by a LUIC augmented event structure.

Restatement of Theorem 1 (Semantic Completeness). *Let $\mathbf{E} = \langle E, \rightsquigarrow, <, d \rangle$ be a LUIC augmented event structure. Then there exist (infinitely many) system evolutions $\mathcal{S}$ that induce $\mathbf{E}$.*

Proof. By the *computation immediacy*, the relation $\rightsquigarrow \cup \dashrightarrow$ is acyclic on E. Thus, there exists at least one ordering of $E = \langle \epsilon_1, \ldots, \epsilon_\ell \rangle$ compatible with $\rightsquigarrow$ and $\dashrightarrow$, i.e. such that $\epsilon_i \rightsquigarrow \epsilon_j$ or $\epsilon_i \dashrightarrow \epsilon_j$ implies $i < j$. Define by induction a system evolution $\mathcal{S}_i$ for $i \leq \ell$ translating the elements of E (in order), starting from the empty system evolution without transitions $\mathcal{S}_0 = \langle \emptyset, \emptyset; \emptyset \rangle$.

Consider a step $i \leq \ell$, let $\delta_i = d(\epsilon_i)$, and add the following three transitions to the system $\mathcal{S}_i = \mathcal{S}_{i-1} \xrightarrow{env} N' \xrightarrow{\delta_i -} N'' \xrightarrow{env} N'''$:

- first, an *env* transition changing the topology to any neighbouring relation $\rightarrowtail$ such that $\{\delta' \mid \delta_i \rightarrowtail \delta'\} = \{d(\epsilon') \mid \epsilon_i \rightsquigarrow \epsilon'\}$;
- secondly, a δ_i transition representing the computation, where the filter F clears out from the value-tree environment $\Psi(\delta_i)$ the value trees corresponding to devices not in $X = \{d(\epsilon') \mid \epsilon' \rightsquigarrow \epsilon_i\}$;
- finally, another *env* transition, which removes δ_i from the domain of the system configuration if $\text{next}(\epsilon_i)$ does not exist, or it does nothing if $\text{next}(\epsilon_i)$ exists.

Then, the system evolution $\mathcal{S}_\ell$ induces $\mathbf{E}$ (c.f. Definition 13). Notice that many system evolutions may induce $\mathbf{E}$: besides the existence of many different linearisations of E according to $\rightsquigarrow$ and $\dashrightarrow$, *env* transitions can be added in an unbounded number of ways. □

Appendix B.2. Proof of Theorem 2

In Appendix A.2 we modelled the message passing resulting from a fire (rule [N-FIR]) as a broadcast of whole value-trees θ. However, only part of that data is actually used in computation, and practical implementations of the calculus (Protelis (Pianini et al., 2015) and ScaFi (Casadei and Viroli, 2016)) take profit of that for greatly reducing the amount of data exchanged. In particular, an optimised implementation may:

- store only values of nodes corresponding to [E-NBR] and [E-SHARE] statements;

- label each of them with the sequence of Boolean results of `if` guards encompassed to reach them (as these are the only values needed to perform alignment).

In measuring the message size required for computations, we consider the above optimised implementation as reference.

Restatement of Theorem 2 (Lightweightness). *The translation* `P` *of a formula* ϕ *according to Figure 6 computes in each node using message size* $O(S)$ *and computation time/space* $O(L + SN)$*, where* N *is neighbourhood size and* L, S *are the numbers of logical and spatial operators in* ϕ*.*

Proof. We proceed by syntactic induction on ϕ. Logical operators are translated into Boolean operations that perform in constant time locally without message exchanges. Thus we only need to prove that spatial operators are translated into programs using $O(1)$ message size and $O(N)$ computation time/space.

Each spatial operator is expanded into a formula with at most four logical operators, at most two occurrences of local operators $\Box$ and $\Diamond$, and at most one occurrence of global operators $\mathcal{F}$ or $\mathcal{R}$. Each occurrence of $\Box$ and $\Diamond$ is translated in a program sending one bit with **`nbr`**`{F}` which scans the data of the N neighbours checking if some (all) is true (thus in $O(N)$ time/space).

Occurrences of $\mathcal{F}$ are translated into a call to **`somewhere`**, triggering an execution of the **`distanceTo`** building block, which exchanges with neighbours a single positive integer (with value up to `D`, which we consider to fit within one word), and selects the minimum from the received values in $O(N)$ time/space.

Finally, occurrences of $\mathcal{R}$ are translated as an **`if`** statement (creating a new non-trivial node in the value-trees, for additional $O(1)$ message size for alignment purposes), together with a call to **`somewhere`** (previously discussed). $\square$

Appendix B.3. Proof of Theorem 3

In this section, we prove that the translation of an SLCS formula into a field calculus monitor is correct and optimal. In order to prove that the translated program is correct and optimal in stabilisation speed, we first need to inspect the convergence properties of the **`distanceTo`** algorithm (Lemma 4, an extended version of results in (Viroli et al., 2018)).

Lemma 4 (Distance-To Stabilisation)**.** *Assume that* ***`distanceTo`***`(dest)` *is computed in a stable and connected network, and let* $d(\delta)$ *be the hop-count distance of a device* δ *in the network to the closest device where* `dest` *holds (*∞ *if no such device exists).*

Then after n *full rounds of execution, devices such that* $d(\delta) < n$ *stabilise to* $d(\delta)$*, while devices such that* $d(\delta) \geq n$ *satisfy* ***`distanceTo`***`(dest) >= n`*.*

Proof. Let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \dots$ be a (possibly infinite) network evolution where the topology and atomic proposition `source` are stable. Assume that N_0 is a configuration attainable from the execution of **`distanceTo`**, in particular, that the values for `d` shared between neighbours are non-negative integers. Let t_n be

such that n full rounds of execution has passed at step t_n since start (so that $t_0 = 0$), and proceed by induction on n.

From the first full round of execution on, each destination device will correctly compute 0 as result of `distanceTo(dest)`, while non-destination devices will compute `minHood(d)+1`. Since values `d` shared between neighbours are always non-negative, the result has to be ≥ 1, completing the proof for $n = 1$.

Assume now that the thesis holds for $n - 1$. Consider a device δ with $d(\delta) = n - 1$, which then has (at least) one neighbour with distance $n - 2$, and no neighbour with distance $< n - 2$ (by definition of hop-count distance). From t_{n-1} on, neighbours with distance $n - 2$ will have already stabilised to $n - 2$, while neighbours with distance $\geq n - 1$ will have computed results $\geq n - 1$ (by inductive hypothesis). It follows that `minHood(d)+1` has to be $n - 1$ for δ from its first fire after t_{n-1} on, concluding this part of the proof.

Consider now a device δ with $d(\delta) \geq n$, which then has no neighbour with distance $< n - 1$ (by definition of hop-count distance).

From t_{n-1} on, all those neighbours will have computed results $\geq n - 1$ (by inductive hypothesis). It follows that `minHood(d)+1` has to be at least n for δ from its first fire after t_{n-1} on, concluding the proof. □

Restatement of Theorem 3 (Self-Stabilisation, Correctness, Optimality). *Let* `P` *be the translation of ϕ according to Figure 6. If the network configuration and atomic propositions stabilise, the result of* `P` *also stabilises to the interpretation of ϕ in that final configuration (regardless of the evolution history of the network). Furthermore, the time required for stabilisation is as small as possible, meaning that no correct inductive translation can stabilise with a smaller worst case of full rounds of execution).*

Proof. We proceed by syntactic induction on formulas, assuming derived operators to be already expanded into the basic ones. As before, let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \dots$ be a (possibly infinite) network evolution where the topology and atomic propositions are stable, assuming that N_0 is attainable from the execution of `P`. Let t_0 be minimum such that results of sub-formula have necessarily stabilised after t_0, and let t_n be such that n full rounds of execution has passed at step t_n since t_0. A formula ϕ can be:

- An **atomic formula** ($\top$, $\bot$, q), which is stable from $t = 0$ since atomic propositions are stable (inductive base case).
- A **logical operator**, whose translation is easily checked to be correct and stable since t_0 when the sub-formulas are stable.
- A **local operator** ($\Box\,\phi_1$ or $\Diamond\,\phi_1$), whose overall translation will stabilise at t_1 after each device performed an additional firing to share the stabilised argument result with neighbours through the `nbr{F1}` construct. The correctness of the result can then be easily checked: e.g., $\Box\,\phi_1$ holds on points where all neighbours satisfy ϕ_1, as the translation `allHoodPlusSelf(nbr{F1})` which holds on devices where all the values received from neighbours for `F1` are `true`. Furthermore, the one-round delay is necessary,

as the sub-formula values from neighbours are needed for computing the overall result, and are not available before t_1.

- A **reaches operator** $\phi_1 \mathcal{R} \phi_2$. In the area where ϕ_1 stabilises to `false`, the overall result simultaneously stabilises to `false` which is the correct result (achieved at t_0 with minimal, zero delay). Due to the properties of the **`if`** construct, the area where ϕ_1 is `true` computes its result in isolation, as if the devices in the complementary area where not present. In particular, each connected component of the `true` area performs its computation independently from the others.

  Consider a connected component where at least one device satisfies ϕ_2 (since stabilisation at t_0), in which the correct result of $\phi_1 \mathcal{R} \phi_2$ would then be `true`. Let $d(\delta)$ be the hop-count distance of a device δ in the area to the closest device where ϕ_2 holds. By Lemma 4, the result of **`distanceTo`**`(F2)` on each device δ stabilises to $d(\delta)$ after $t_{d(\delta)}$, and the result of **`somewhere`**`(F2)` (hence **`reaches`**`(F1, F2)`) on each device δ stabilises to `true` after $t_{d(\delta)}$ as well. Furthermore, no correct program could stabilise before $t_{d(\delta)}$, since the information that a device area is a source travels one hop at a time, thus is not available in δ before $t_{d(\delta)}$.

  Finally, consider a connected component where no device satisfies ϕ_2 (since stabilisation at t_0), in which the correct result of $\phi_1 \mathcal{R} \phi_2$ would then be `false`. By Lemma 4, the result of **`distanceTo`**`(F2)` on each device is $\geq n$ after t_n. In particular, after $t_{\mathtt{D}+1}$ we will have **`distanceTo`**`(F2) > D` so that **`somewhere`**`(F2)` and **`reaches`**`(F1, F2)` stabilise to `false`. In fact, no correct program could stabilise before $t_{\mathtt{D}+1}$ in all cases, since the information of a (plausible) point satisfying ϕ_2 at distance `D` would not be available before that moment.

- A **somewhere operator** $\mathcal{F} \phi_1$. In that case, correctness and optimality follow by the same reasoning as $\mathcal{R}$, through the equivalent formula $\top \mathcal{R} \phi_1$. □