Real-Time Guarantees for SLCS Monitors in XC

Giorgio Audrito
Dipartimento di Informatica,
Università degli Studi di Torino
Turin, Italy
giorgio.audrito@unito.it

Ferruccio Damiani
Dipartimento di Informatica,
Università degli Studi di Torino
Turin, Italy
ferruccio.damiani@unito.it

Gianluca Torta
Dipartimento di Informatica,
Università degli Studi di Torino
Turin, Italy
gianluca.torta@unito.it

Abstract

The behavior of distributed systems situated in space can be required to satisfy spatial properties, in addition to the more widely known temporal properties. In particular, it has been previously shown that fully distributed monitors in eXchange Calculus (XC) can be automatically derived for verifying properties of situated systems expressed in the Spatial Logic of Closure Spaces (SLCS). While it has been proven that such monitors eventually compute the truth value of the desired properties, the actual time required for such computations has been thus far disregarded. In the present paper, we fill this gap by investigating the real-time guarantees that can be given in terms of upper bounds on the time taken by the XC monitors to compute the truth of SLCS properties after stabilisation of inputs.

CCS Concepts

• **Computing methodologies** → **Distributed programming languages**; • **Theory of computation** → **Modal and temporal logics**.

Keywords

runtime verification, aggregate computing, spatial logic, real-time

ACM Reference Format:
Giorgio Audrito, Ferruccio Damiani, and Gianluca Torta. 2024. Real-Time Guarantees for SLCS Monitors in XC. In *Proceedings of the 7th ACM International Workshop on Verification and Monitoring at Runtime Execution (VORTEX '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3679008.3685545

1 Introduction

Runtime monitoring is a lightweight verification technique dealing with the observation of a system execution with respect to a specification [22]. Specifications are usually trace- or stream-based, with events mapped to atomic propositions in the underlying logic of the specification language, e.g., *Linear Time Logic (LTL)* [20, 23], or *LTL on finite traces (LTLf)* [25].

In this paper we consider distributed runtime monitoring, for monitoring distributed (situated) systems using distributed monitors. Distribution is particularly challenging for verification purposes, as it requires to deal with issues such as synchronisation, communication faults, lack of global time, and so on. Following Francalanza et al's terminology [19], we assume that the nodes of the distributed system are (mutually) *remote* processes that can dynamically appear and disappear, each one producing a *local trace of events*, i.e., a sequence of sets of observable values derived from the node's sensors or behaviour. Monitors check properties of the system by analysing their traces. We consider an *online* evaluation strategy, where the monitors are executed together with the processes themselves, being hosted at the same locations and communicating with neighbour monitors.

In particular, we focus on fully distributed and decentralised runtime monitors [19] expressed in the *eXchange Calculus (XC)* [4, 5], which have been shown to be automatically derivable from formulas expressed in the *Past Computation Tree Logic (Past-CTL)* and the *Spatial Logic of Closure Spaces (SLCS)* [6, 8]. Such monitors are able to deal with all the assumptions mentioned above (full decentralization, dynamic network, situated nodes), and eventually compute the truth value of the desired properties. However, up to now the actual time required for computing such truth values, and to update them when they change, has been disregarded.

To fill this gap, in this paper we investigate the real-time guarantees that can be given in terms of upper bounds on the time taken by the XC monitors to compute the truth of SLCS properties after stabilisation of the monitored system inputs (i.e., once the value of the sensors on each device and the communication topology of the network stop changing). This is particularly important for SLCS, since the truth of its formulas on a given node can depend on the (local estimate) of the *current* status of remote nodes, and we would like such estimate to be as up-to-date as possible (see Section 3). Interestingly, we will be able to show that the correct truth value of SLCS properties can be evaluated by XC monitors with a delay which linearly depends on the syntactic complexity of the monitored property, the maximum hop distance among system nodes, and the inverse of the monitoring frequency.

2 eXchange Calculus

Aggregate Computing [14] has emerged as a generalization of various previous approaches to programming ensembles of devices [13, 26], applicable to distributed applications deployed in far edge, fog or cloud environments [9], and general multi-agent systems [7]. A central feature of this programming model is its ability to express complex distributed processes through function composition, with operational semantics essentially reduced to asynchronously exchanging values with neighbours. Drawing inspiration from "fields" in physics, this is accomplished through the concept of a computational field, which is defined as a global data structure that maps devices in the distributed system to computational values. These fields can be computed from a set of input fields (such as those from

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

VORTEX '24, September 19, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1119-0/24/09
https://doi.org/10.1145/3679008.3685545

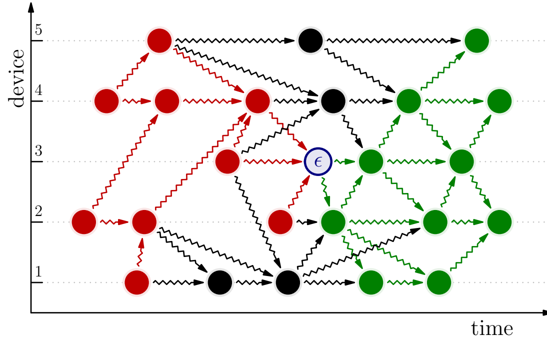

Figure 1: A sample event structure, split in events ϵ' in the causal past of ϵ ($\epsilon' < \epsilon$, in red), events in the causal future ($\epsilon < \epsilon'$, in green) and concurrent (non-ordered, in black).

sensors) either at a low level using simple programming language constructs or at a high level by composing general-purpose building blocks of reusable behavior. Ultimately, output fields can be utilized by actuators.

The reference programming models for aggregate computing, i.e. the *Field Calculus (FC)* [14, 26] and its recently proposed extension the *eXchange Calculus (XC)* [4, 5], are both implemented by the *Field Calculus++ (FCPP)* C++ library [2, 11, 12] and the *Scala Fields (ScaFi)* [15–17] Scala library. They are core universal [3] languages for aggregate computations over distributed networks of (possibly) mobile devices, each capable of asynchronously performing simple local computations, and of interacting with a neighbourhood by local exchanges of messages. XC provides mechanisms to express and compose such distributed computations, on a level of abstraction that avoids the explicit management of message exchanges, device position and quantity, and so on. In this context, a single program is periodically and asynchronously executed on every device, according to a cyclic schedule of *rounds*. In each round, the device gathers sensors' inputs and recently collected messages; evaluates the program; and broadcasts the result to neighbours and (possibly) actuators. Through the repeated execution of rounds across space (where devices are located) and time (when devices initiate a new cycle), a global behavior emerges. This behavior can be understood as occurring on the overall network of interconnected devices, modeled as a single aggregate machine with a neighboring relation. A formal semantics is provided to XC programs through the classical notion of *event structure* [21], which is also used to interpret temporal logic formulas. An *event structure* is a finite set of events E along with an acyclic *neighbouring relation* $\rightsquigarrow \subseteq E \times E$ modelling message passing. A sequence of neighbour events $\epsilon_1 \rightsquigarrow \ldots \rightsquigarrow \epsilon_n$ is referred to as a *message path*.

Event neighboring induces *causality relation* $< \subseteq E \times E$, defined as the transitive closure of $\rightsquigarrow$ and modeling causal dependence. An example structure is illustrated in Figure 1. In practice, event structures arise from device neighborhood graphs changing over time. For instance, in Figure 1, device 3 appears at a certain point in time with devices 4 and 1 as neighbors, but after a few steps, its neighbors become devices 2 and 4.

Figure 2 illustrates the syntax of the XC language. It is a core functional language where expression e can be either: a variable x; a (possibly recursive) function `fun` $\mathtt{x}(\overline{\mathtt{x}})$, which may have free variables; a function call $\mathtt{e}_0(\overline{\mathtt{e}})$; a let-like expression `val` $\mathtt{x} = \mathtt{e}_1; \mathtt{e}_2$;

Syntax:

$\mathtt{e} ::= \mathtt{x} \mid \texttt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_0\} \mid \mathtt{e}_0(\overline{\mathtt{e}}) \mid \texttt{val}\ \mathtt{x} = \mathtt{e}_1; \mathtt{e}_2 \mid \ell \mid \mathtt{w}$	expression
$\mathtt{w} ::= \ell[\overline{\delta} \mapsto \overline{\ell}]$	nvalue
$\ell ::= \mathtt{b} \mid \texttt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_0\} \mid \mathtt{c}(\overline{\ell})$	local literal
$\mathtt{b} ::= \mathtt{s} \mid \texttt{exchange} \mid \texttt{nfold} \mid \texttt{mux} \mid \ldots$	built-in function

Syntactic sugar:

$(\overline{\mathtt{x}})\texttt{=>}\mathtt{e}$	$::= \texttt{fun}\ \mathtt{y}(\overline{\mathtt{x}})\{\mathtt{e}\}$ *where* y *is a fresh variable*
$\texttt{def}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\}$	$::= \texttt{val}\ \mathtt{x} = \texttt{fun}\ \mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}\};$
$\texttt{if}(\mathtt{e})\{\mathtt{e}_\top\}\ \texttt{else}\ \{\mathtt{e}_\bot\}$	$::= \texttt{mux}(\mathtt{e}, ()\texttt{=>}\mathtt{e}_\top, ()\texttt{=>}\mathtt{e}_\bot)()$

Figure 2: Syntax of the eXchange Calculus (XC) language.

a local literal ℓ, that is either a built-in function b, a defined function `fun` $\mathtt{x}(\overline{\mathtt{x}})\{\mathtt{e}_0\}$ without free variables, or a data constructor c applied to local literals (possibly none); a *neighbouring value* w, which represents a (set of) local literals either received from or sent to neighbouring devices. These values are defined as maps $\mathtt{w} = \ell[\delta_1 \mapsto \ell_1, \ldots, \delta_n \mapsto \ell_n]$, mapping device identifiers δ_i to their respective local literals ℓ_i. Additionally, there is a local literal ℓ that serves as a *default*. The language models local values ℓ (those not dependent on neighbours) as a special case of neighbouring values $\ell[]$, containing only the default local literal. Thus, in XC each value can be considered as a neighbouring value. Standard built-in operations (such as sum and product) operate on neighbouring values in a point-wise manner (device by device). Neighbouring values can be aggregated into a single local value through a functional-style fold operation $\texttt{nfold}(\mathtt{f}, \mathtt{w}, \ell)$, of type[1] $\forall T.\ ((T, T) \to T, T, T) \to T$, which applies a specified binary operation f (of type $(T, T) \to T$) repeatedly to the local values that form w (of type T), except for the current device δ, whose value is taken from the third argument.

In XC, evaluation is performed within a context that includes all messages received from neighboring devices. A process known as *alignment* ensures that for each sub-expression, the context is limited to values corresponding to the same sub-expression on neighboring devices. These context values are primarily utilized by a built-in function called *exchange*, which models communication and whose semantics is described below. The expression:

```
exchange(e0, (x) => return er send es)
```

where **return** $\mathtt{e}_r$ **send** $\mathtt{e}_s$ is syntactic sugar for pair construction $(\mathtt{e}_r, \mathtt{e}_s)$, is evaluated according to the following steps;

- First, gather a *neighbouring* value n (of a type T) that maps each neighbor δ' to the last value shared by δ' for this exchange expression.
- If it is the first execution of **exchange** on the current device δ, the value of $\mathtt{e}_0$ (also of type T) is used as the value for δ in n. Otherwise, the value shared by δ itself in its previous round is used instead.
- Next, evaluate $\mathtt{e}_r$ by substituting n for x, resulting in a value $\mathtt{v}_r$ (of a possibly different type R). Do the same for $\mathtt{e}_s$ obtaining value $\mathtt{v}_s$ (which is also of type T).
- Finally, return $\mathtt{v}_r$ as the value of the **exchange** expression in the program, and broadcast $\mathtt{v}_s$ to neighbors, who will use it in their subsequent rounds to produce their neighboring value n.

[1] Since in XC every value is treated as a neighbouring value, the type system does not distinguish between local and neighbouring values as well, assigning to both the same types T – see [5] for details.

$$\begin{array}{llr} \phi,\psi ::= & \bot \,|\, \top \,|\, q \,|\, (\neg\phi) \,|\, (\phi\wedge\psi) \,|\, (\phi\vee\psi) \,|\, (\phi\Rightarrow\psi) \,|\, (\phi\Leftrightarrow\psi) & \text{prop.} \\ & |\, (\Box\,\phi) \,|\, (\Diamond\,\phi) \,|\, (\partial\,\phi) \,|\, (\partial^-\,\phi) \,|\, (\partial^+\,\phi) & \text{spatial} \\ & |\, (\phi\,\mathcal{R}\,\psi) \,|\, (\phi\,\mathcal{T}\,\psi) \,|\, (\phi\,\mathcal{U}\,\psi) \,|\, (\mathcal{G}\,\phi) \,|\, (\mathcal{F}\,\phi) & \end{array}$$

Figure 2: Syntax of SLCS (as in [6, 18]).[2]

Thus, overall the exchange function has type $\forall T.\forall R.\ (T, (T) \to (R, T)) \to R$. Consider, as an example, the following function declaration:

```
def dist(source) { // of type (bool) -> num
  exchange(
    infinity,
    (d) => retsend mux(source, 0, nfold(min, d, infinity)+1)
  )
}
```

In the function declaration above, we use **retsend** e as syntactic sugar for **return** e **send** e. Function `mux(b,x,y)` is the multiplexer function returning either x or y depending on whether b is true or false, respectively. The `dist` function calculates hop-count distances from the nearest device where `source` is true, using a single **exchange** construct. This construct collects distance estimates from neighboring devices into a neighboring value n (which maps neighbor devices to their distance estimates, defaulting to ∞ when no information is available), and assigns it to the variable d. The inner anonymous function body returns zero for source devices. For other devices, as in the traditional Bellman-Ford algorithm, it returns the minimum value from the neighbors' estimates `nfold(min, d, infinity)` (thus excluding the current device, for which `infinity` is used) increased by one.

In the next section, we will also use function **nbr** for convenience, that can be defined through **exchange** as follows:

```
def nbr(e0, e) { exchange(e0, (n) => return n send e) }
```

The **nbr** function returns the neighbouring value n, while it sends the value of expression e to all the neighbours (including the current device), and has type $\forall T.\ (T, T) \to T$.

3 Spatial Logic of Closure Spaces

Figure 2 presents the syntax of the *Spatial Logic of Closure Spaces (SLCS)* [6, 18]. It is based on atomic propositions q representing observables, that can be combined with usual propositional logic operators, and further enriched with spatial modalities, that can be divided into *local* and *global*. The local modalities are:

- $\Box\,\phi$ (where $\Box$ is the *interior* operator): true at points where all neighbours satisfy ϕ (intuitively, the area in which ϕ is true reduced on its border);
- $\Diamond\,\phi$ (where $\Diamond$ is the *closure* operator): true at points where a neighbour satisfies ϕ (intuitively, the area in which ϕ is true enlarged on its border);
- $\partial\,\phi$ (where ∂ is the *boundary* operator): true at points where $\Diamond\,\phi$ holds and $\Box\,\phi$ does not hold (intuitively, points that sit at the border of the area in which ϕ is true);
- $\partial^-\,\phi$ (where ∂^- is the *interior boundary* operator): true at points where formula ϕ holds and formula $\Box\,\phi$ does not hold (intuitively, the part of the border that is also in the area);
- $\partial^+\,\phi$ (where ∂^+ is the *closure boundary* operator): true at points where formula $\Diamond\,\phi$ holds and formula ϕ does not hold) (intuitively, the part of the border that is not in the area).

We choose $\Diamond$ as primitive, expressing the others through the equivalences[3] $\Box\,\phi \triangleq \neg\Diamond\neg\phi$, $\partial\,\phi \triangleq (\Diamond\,\phi)\wedge\neg(\Box\,\phi)$, $\partial^-\,\phi \triangleq \phi\wedge\neg(\Box\,\phi)$, $\partial^+\,\phi \triangleq (\Diamond\,\phi)\wedge\neg\phi$. The global modalities are:

- $\phi\,\mathcal{R}\,\psi$ (where $\mathcal{R}$ is the *reaches* operator): true at the ending points of paths in the graph whose starting point satisfies ψ, and where ϕ holds on each point on the path (intuitively, points that are able to reach the area where ψ is true while staying in the area where ϕ is true);
- $\phi\,\mathcal{T}\,\psi$ (where $\mathcal{T}$ is the *touches* operator): true at the end of paths whose start satisfies ψ and where ϕ holds in the rest of the path (so ϕ may not hold at the start of the path);
- $\phi\,\mathcal{U}\,\psi$ (where $\mathcal{U}$ is the *surrounded by* operator): true at points in an area satisfying ϕ, whose closure boundary satisfies ψ;
- $\mathcal{G}\,\phi$ (where $\mathcal{G}$ is the *everywhere* operator): true where ϕ holds in every point of every incoming path;
- $\mathcal{F}\,\phi$ (where $\mathcal{F}$ is the *somewhere* operator): true where ϕ holds in some point of some incoming path.

We choose $\mathcal{R}$ as primitive, expressing the others through $\phi\,\mathcal{T}\,\psi \triangleq \phi\,\mathcal{R}\,\Diamond\,\psi$, $\phi\,\mathcal{U}\,\psi \triangleq \phi\wedge\Box\neg(\neg\psi\,\mathcal{R}\,\neg\phi)$, $\mathcal{F}\,\phi \triangleq \top\,\mathcal{R}\,\phi$, $\mathcal{G}\,\phi \triangleq \neg\mathcal{F}\,\neg\phi$.

Example 3.1. As an example, consider the statement *"dangerous* (**d**) *areas are surrounded by devices who can reach safely a base* (**b**)*"*:

$$\phi \coloneqq \mathbf{d} \Rightarrow (\mathbf{d}\,\mathcal{U}(\neg\mathbf{d}\,\mathcal{R}\,\mathbf{b}))$$

Analysing the formula bottom-up, in sub-formula $\phi_1 \coloneqq (\neg\mathbf{d}\,\mathcal{R}\,\mathbf{b})$ we use modality $\mathcal{R}$ to select devices that can reach a base **b** without crossing dangerous places **d**. Then, in sub-formula $\phi_2 = (\mathbf{d}\,\mathcal{U}\,\phi_1)$, we use modality $\mathcal{U}$ to select devices belonging to areas where **d** holds, while the devices in their closure boundaries satisfy ϕ_1. Finally, in the complete formula $\phi \coloneqq \mathbf{d} \Rightarrow \phi_2$, we use the $\Rightarrow$ operator to express the fact that ϕ_2 must hold wherever there is danger **d**.

As a second example, consider the statement *"while staying next to secure* (**s**) *points, it is possible to approach an unsecured area surrounded by danger* (**d**)*"*:

$$\phi \coloneqq \Diamond\,\mathbf{s}\,\mathcal{T}(\neg\mathbf{s}\,\mathcal{U}\,\mathbf{d})$$

Again, moving bottom-up, in sub-formula $\phi_1 \coloneqq \Diamond\,\mathbf{s}$ we select devices next to a secure point (where **s** holds). In sub-formula $\phi_2 \coloneqq (\neg\mathbf{s}\,\mathcal{U}\,\mathbf{d})$ we use modality $\mathcal{U}$ to select devices belonging to insecure areas surrounded by dangerous points (where **d** holds). Finally, in the complete formula $\phi \coloneqq \phi_1\,\mathcal{T}\,\phi_2$ we use modality $\mathcal{T}$ to select devices that can reach a place where ϕ_2 holds through a path where ϕ_1 holds everywhere (except possibly in the target place).

The translation of SLCS formulas into XC introduced in [6] is shown in Figure 3, by recursion on sub-formulas. We translate the truth value $\top$ into the XC literal `true`, atomic propositions q into built-in function calls `q()` getting their value from some external environment, and propositional logic operators into their

[2]Note that the syntax of SLCS is heavily inspired by temporal logics such as LTL, but it associates different meaning to identically written modalities, as it is a *spatial* rather than *temporal* logic. The syntax we use for SLCS is the standard one in literature [6, 18] and its informal meaning is detailed in Section 3.

[3]The symbol $\triangleq$ means "defined as".

$[\![\top]\!]$	= `true`	$[\![q]\!]$	= `q()`
$[\![\neg\phi]\!]$	= `!`$[\![\phi]\!]$	$[\![\phi \vee \psi]\!]$	= $[\![\phi]\!]$ `||` $[\![\psi]\!]$
$[\![\Diamond\,\phi]\!]$	= `nfold(||, nbr(false,` $[\![\phi]\!]$`))`		
$[\![\phi\,\mathcal{R}\,\psi]\!]$	= `if (`$[\![\phi]\!]$`) {dist(`$[\![\psi]\!]$`)<D} else {false}`		

Figure 3: Translation of a primitive set of SLCS operators into XC [6]. The translation of a formula ϕ is denoted by $[\![\phi]\!]$.

XC representation. In the translation of the local modality $\Diamond$ and global modality $\mathcal{R}$ we assume that:

- `nbr` and `dist` are as in Section 2. We consider an overload of `nfold` without the third argument, hence using the second argument also for the current device;
- `D` is an integer number providing an upper bound to the network hop-count diameter (either fixed at design time or estimated through an XC expression).

In particular, the translation of $\Diamond\,\phi$ first requires to receive the value of (the XC translation of) ϕ from the neighbours, using the `nbr` function defined above. Note that the current device is initialized as false. Applying `nfold(||, _)` to n gives true iff the value of ϕ is `true` in at least one of the neighbours (or in the device itself), which is the definition of the closure modality $\Diamond$.

Considering the translation of $\phi\,\mathcal{R}\,\psi$ notice first that, as function `dist` is computed within a branch, it only receives messages from neighbours which selected the same branch, i.e., for which ϕ is true. The sources for the computation of `dist` are all the devices where ψ is true, thus `dist` will eventually compute the distance from the closest device where ψ holds. If the value returned by `dist` is larger than the network diameter `D`, it means that it is either a transient incorrect value, or that it is impossible to reach a device where ψ holds. In either case, the whole formula $\phi\,\mathcal{R}\,\psi$ is considered false. Otherwise, it is a plausible hop-count distance from the current device to a device where ψ holds, considering only devices where ϕ holds. Therefore $\phi\,\mathcal{R}\,\psi$ is considered true.

Example 3.2. Let us consider the first formula in Example 3.1. Sub-formula $\phi_1 := (\neg\mathbf{d}\,\mathcal{R}\,\mathbf{b})$ is translated to:

```
if (!d) {dist(b) < D} else {false}
```

Sub-formula $\phi_2 = (\mathbf{d}\,\mathcal{U}\,\phi_1)$ can be rewritten as:

$$(\mathbf{d} \wedge \Box\,\neg(\neg\phi_1\,\mathcal{R}\,\neg\mathbf{d})) = (\mathbf{d} \wedge \neg\,\Diamond(\neg\phi_1\,\mathcal{R}\,\neg\mathbf{d}))$$

using the definition of $\mathcal{U}$ in terms of primitive modalities. This translates to:

```
d && !nfold(||, nbr(false,
                   if (!ϕ1) {dist(!d) < D} else {false}))
```

4 Real-Time Analysis

Given that the SLCS logic is evaluated on static graphs, an SLCS monitor cannot precisely determine the value of its formula in real-time, if the underlying graph can change over time. However, the XC translation has been proven to be *self-stabilizing* to the desired monitor output. This means it will eventually converge to the correct output, assuming that the network graph and atomic propositions remain unchanged for a *sufficiently long* period. In order to properly characterize the performance of the XC translation, it is thus crucial to bound this period after which the monitor output coincides with the translated formula.

We follow the approach delineated in [10] to characterise real-time behaviour of distributed programs, and specifically XC programs. In particular, a distributed program can be abstracted to a mathematical function:

$$f : [\![T_i]\!]_{\mathbf{ST}} \rightharpoonup [\![T_o]\!]_{\mathbf{ST}}$$

where **ST** denotes the fact that the types of the inputs/outputs of f contain values depending on space and time. Details on the mathematical definition of this notation can be found in [26, Sec. 3.3], which provides a denotational semantics for aggregate computing based on event structures. Each monitor derived from an SLCS formula is a function `f` in the **SD-TI** class, meaning that it is *discretely dependent* on space (**D**), and *independent* (**TI**) of time (in the sense that it is self-stabilizing as discussed above). According to [10], then, the input and output types $[\![T_i]\!]_{\mathbf{ST}}$ and $[\![T_o]\!]_{\mathbf{ST}}$ of the abstraction f are graphs with values associated to each node, i.e., $\mathcal{G} \to [\![T]\!]$. The graphs $\mathcal{G}$ correspond to the topology of the network, i.e., they are directed graphs $\langle\{\delta_i\}, \rightarrowtail\rangle$ on the set of network devices $\{\delta_i\}$ (note the difference w.r.t. to the graph of an Event Structure such as the one in Figure 1). We aim to find a real-time constraint specification in the following **Minimal** form:

Specification 1 (Minimal). *(adapted from [10]) Assume that after a certain time t_0, inputs and topology of a distributed monitor of SLCS formula ϕ are fixed. Then, after time $t_0 + \Delta(\phi)$ for a given $\Delta(\phi)$, the truth value computed by the monitor corresponds exactly to that of ϕ.*

In the following sections we'll derive a function Δ mapping SLCS formulas ϕ to a real value $\Delta(\phi)$ that satisfies the specification above, given the following additional assumptions:

- Every connected subset of the network always has a diameter (in hops) lower than a given parameter D;
- Every device performs a round of computation at least every τ seconds.

Note that the granularity of time considered in our analysis is thus the maximum round period τ.

4.1 The Reachability Operator

First, let us analyse the behaviour of the main SLCS operator, that is, the reachability operator $\phi_1\,\mathcal{R}\,\phi_2$. Assume that after time t_0, the truth values of ϕ_1 and ϕ_2 computed by the distributed monitor have stabilised in each device δ. Consider separately the following three kinds of regions of the network graph $\langle\{\delta_i\}, \rightarrowtail\rangle$, also depicted in Figure 4 for a sample scenario.

(1) *Devices for which ϕ_1 is* false: they correctly compute the value *false* from t_0 on, without further delay.
(2) *Devices for which ϕ_1 is* true*, but no path consisting of devices for which ϕ_1 is* true *and starting in a device for which ϕ_2 is* true *exists*: in this case, the value of $\phi_1\,\mathcal{R}\,\phi_2$ should be false. However, at time t_0, the values of `dist(`ϕ_2`)` computed by the various devices could be any non-negative integer. Consider a (weakly) connected region of such devices, and notice that each of them has neighbours that are either in the same region, or such that ϕ_1 is *false*. Between time t_0 and $t_0 + \tau$, every device in the region updates its monitor at least once, and since none of them is a source, it will compute a distance ≥ 1. Between time $t_0 + \tau$ and $t_0 + 2\tau$, every device

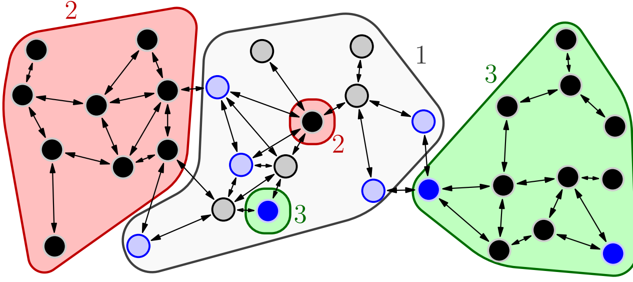

Figure 4: Representation of the truth value of $\phi_1 \mathcal{R} \phi_2$ in a sample network. Nodes for which ϕ_1 holds are full circles, nodes for which it does not are hollow. Nodes for which ϕ_2 holds are blue, nodes for which it does not are black. The areas labelled as 1, 2 and 3 correspond to the three kinds of regions identified in Section 4.1.

in the area will update again its monitor, and it will notice that each neighbour has a distance ≥ 1, concluding that its distance must be ≥ 2. Similarly, by induction, after time $t_0 + n\tau$ every device in the area will have computed a distance $\geq n$. In particular, after time $t_0 + D\tau$ every device will have computed a distance $\geq D$, hence correctly concluding that $\phi_1 \mathcal{R} \phi_2$ is *false*.

(3) *Devices for which ϕ_1 is* true*, and a path consisting of devices for which ϕ_1 is* true *and starting in a device δ for which ϕ_2 is* true *exists*: in this case, `dist(`ϕ_2`)` is zero in δ from time t_0. Then, after time $t_0 + \tau$, each device at 1 hop from δ correctly computes its distance equal to 1. Similarly, by induction, after time $t_0 + n\tau$ each device within n hops computes its distance correctly equal to n. Since the network diameter of a region is by assumption lower than D, after time $t_0 + D\tau$ every device in the region will have computed a distance lower than D, hence correctly concluding that $\phi_1 \mathcal{R} \phi_2$ is *true*.

Thus, in each of those cases, the monitor converges to the correct result within $D\tau$ seconds after the convergence of the sub-formulas:

$$\Delta(\phi_1 \mathcal{R} \phi_2) = \max(\Delta(\phi_1), \Delta(\phi_2)) + D\tau$$

Note that we take the max (instead of the sum) of $\Delta(\phi_1), \Delta(\phi_2)$, because each sub-formula can converge to its final value simultaneously, without ϕ_2 needing to wait for the stabilisation of ϕ_1.

4.2 Compositional Extension to SLCS

We are now able to define $\Delta(\phi)$ for each SLCS formula, by induction on the formula structure. The correctness of the following discussion can be seen informally, by referring to the translation of SLCS formulas described in Section 3 above.

As the base induction case, ϕ could be an atomic proposition or constant $\top/\bot$. In all those cases, the monitor stabilises immediately after t_0, and thus $\Delta(\phi) = 0$.

If $\phi = \neg\phi'$, the monitor stabilises immediately once ϕ' stabilises, thus $\Delta(\phi) = \Delta(\phi')$. If $\phi = \phi_1 \vee \phi_2$, the monitor also stabilises immediately once both sub-formulas stabilise, thus $\Delta(\phi) = \max(\Delta(\phi_1), \Delta(\phi_2))$.

Consider now that $\phi = \Diamond \phi'$. Between $t_0 + \Delta(\phi')$ and $t_0 + \Delta(\phi') + \tau$, every device updates its value, computing whether any of its neighbours evaluated ϕ' to *true*. Since after $t_0 + \Delta(\phi')$ the values of ϕ' are correct in each device, it follows that after $t_0 + \Delta(\phi') + \tau$ the values of ϕ are correct, thus:

$$\Delta(\phi) = \Delta(\phi') + \tau$$

The same reasoning and result applies to every other local modality ($\Box, \partial, \partial^+, \partial^-$).

Recall that if $\phi = \phi \mathcal{R} \phi'$, by the reasoning in the previous subsection, $\Delta(\phi) = \max(\Delta(\phi_1), \Delta(\phi_2)) + D\tau$. Consider now that $\phi = \mathcal{F} \phi'$ (resp. $\mathcal{G} \phi'$). By the equivalences in Section 3, $\phi = \top \mathcal{R} \phi'$ (resp. $\neg(\top \mathcal{R} \neg\phi')$). By applying the results obtained so far:

$$\Delta(\phi) = \max(\Delta(\top), \Delta(\phi')) + D\tau = \Delta(\phi') + D\tau$$

(and the same holds for $\mathcal{G}$, since $\neg$ does not increase the value of Δ).

Finally, consider now that $\phi = \phi_1 \mathcal{T} \phi_2$ or, resp., $\phi_1 \mathcal{U} \phi_2$. By the equivalences provided in Section 3, $\phi = \phi_1 \mathcal{R} \Diamond \phi_2$ or, resp., $\phi_1 \wedge \Box \neg(\neg\phi_2 \mathcal{R} \neg\phi_1)$. In the former case:

$$\Delta(\phi) = \max(\Delta(\phi_1), \Delta(\phi_2) + \tau) + D\tau$$

because an extra round is needed for the truth of $\Diamond \phi_2$. In the latter case:

$$\Delta(\phi) = \max(\Delta(\phi_1), \Delta(\phi_2)) + (D + 1)\tau$$

because an extra final round is needed for the truth of $\Box \neg(\ldots)$.

By combining all these relations together, we can derive $\Delta(\phi)$ for each SLCS formula ϕ. Note that $\Delta(\phi)$ grows at most by $(D+1)\tau$ for each level of complexity or *depth*[4] of ϕ. Since useful formulas typically have small depths, they can be evaluated efficiently provided the network diameter is not too large and/or the device rounds are scheduled at a sufficiently high frequency.

Example 4.1. Considering the first formula in Example 3.1, we notice that $\Delta(\neg\mathbf{d} \mathcal{R} \mathbf{b}) = \max(\Delta(\neg\mathbf{d}), \Delta(\mathbf{b})) + D\tau = \max(0, 0) + D\tau = D\tau$. Since "$\Rightarrow$" is a logical operator, it does not increase the value of Δ, and thus $\Delta(\phi)$ is:

$$\max(\Delta(\mathbf{d}), \Delta(\mathbf{d}\, \mathcal{U} (\neg\mathbf{d} \mathcal{R} \mathbf{b}))) = \max(0, \max(0, D\tau) + D\tau) = 2D\tau$$

Let us now consider the second formula in Example 3.1. First, $\Delta(\neg\mathbf{s}\, \mathcal{U}\, \mathbf{d}) = \max(0, 0) + (D + 1)\tau = (D + 1)\tau$. Then:

$$\Delta(\phi) = \max(\tau, (D + 1)\tau + \tau) + D\tau = 2(D + 1)\tau$$

5 Conclusions

In this paper, we explored the real-time guarantees achievable for XC monitors of SLCS spatial properties, by establishing upper bounds on the time required for output stabilisation after the stabilization of system inputs. This aspect is crucial for SLCS because the truth value of its formulas at a particular node may rely on the (local estimate of the) *current* status of remote nodes, necessitating the most up-to-date estimates. Notably, we demonstrated that XC monitors can evaluate the correct truth value of SLCS properties with a delay that scales linearly with the syntactic complexity of the monitored property, the maximum hop distance between system nodes, and the reciprocal of the monitoring frequency.

The current result is only a starting point for the real-time management of aggregate spatial monitors. In particular, the guarantee

[4]The *depth* of a formula is the longest chain of nested modalities that it contains. For instance, both formulas in Example 3.1 have a depth of 2.

provided only considers the convergence time after input stabilisation, hence it does not apply to scenarios where the input network never stabilises, and instead slowly changes over time. Future work may address this by considering stronger properties, characterising the output error given bounds on the speed at which the input continuously changes. Furthermore, in this paper we only considered the SLCS translation provided in literature [6]. Other translations of the $\mathcal{R}$ operator may be possible and preferable depending on the scenario (as per the preliminary investigation in [1]), and real-time guarantees could be provided for those alternative translations. Finally, extension of the SLCS logic have been proposed so far, such as the *Signal Spatio-Temporal Logic* (SSTL) [24], which enrich the modalities (for example) with metric bounds. Translations into XC with corresponding real-time guarantees could be investigated for such extended logics in future work.

Acknowledgments

This study has been supported by the Italian PRIN project "CommonWears" (2020HCWWLP). It is part of the project NODES, which has received funding from the MUR – M4C2 1.5 of PNRR funded by the European Union - NextGenerationEU (Grant agreement no. ECS00000036), and it was carried out within the Agritech National Research Center and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1032 17/06/2022, CN00000022). This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

[1] Gianluca Aguzzi, Giorgio Audrito, and Mirko Viroli. 2024. Optimising Aggregate Monitors for Spatial Logic of Closure Spaces Properties. In *VORTEX 2024: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution*. ACM. https://doi.org/10.1145/3679008.3685544

[2] Giorgio Audrito. 2020. FCPP: an efficient and extensible Field Calculus framework. In *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 153–159. https://doi.org/10.1109/ACSOS49614.2020.00037

[3] Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Mirko Viroli. 2018. Space-Time Universality of Field Calculus. In *Coordination Models and Languages (COORDINATION) (Lecture Notes in Computer Science, Vol. 10852)*. Springer, 1–20. https://doi.org/10.1007/978-3-319-92408-3_1

[4] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. 2022. Functional Programming for Distributed Systems with XC. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:28. https://doi.org/10.4230/LIPICS.ECOOP.2022.20

[5] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. 2024. The eXchange Calculus (XC): A functional programming language design for distributed collective systems. *J. Syst. Softw.* 210 (2024), 111976. https://doi.org/10.1016/J.JSS.2024.111976

[6] Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Volker Stolz, and Mirko Viroli. 2021. Adaptive distributed monitors of spatial properties for cyber-physical systems. *J. Syst. Softw.* 175 (2021). https://doi.org/10.1016/j.jss.2021.110908

[7] Giorgio Audrito, Roberto Casadei, and Gianluca Torta. 2021. Towards Integration of Multi-Agent Planning with Self-Organising Collective Processes. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2021, Companion Volume, Washington, DC, USA, September 27 - Oct. 1, 2021*, Esam El-Araby, Vana Kalogeraki, Danilo Pianini, Frédéric Lassabe, Barry Porter, Sona Ghahremani, Ingrid Nunes, Mohamed Bakhouya, and Sven Tomforde (Eds.). IEEE, 297–298. https://doi.org/10.1109/ACSOS-C52956.2021.00042

[8] Giorgio Audrito, Ferruccio Damiani, Volker Stolz, Gianluca Torta, and Mirko Viroli. 2022. Distributed runtime verification by past-CTL and the field calculus. *J. Syst. Softw.* 187 (2022). https://doi.org/10.1016/j.jss.2022.111251

[9] Giorgio Audrito, Ferruccio Damiani, and Gianluca Torta. 2022. Bringing Aggregate Programming Towards the Cloud. In *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 13703)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 301–317. https://doi.org/10.1007/978-3-031-19759-8_19

[10] Giorgio Audrito, Ferruccio Damiani, and Gianluca Torta. 2024. Towards Real-Time Aggregate Computing. (2024). 12th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2024). To appear.

[11] Giorgio Audrito, Luigi Rapetta, and Gianluca Torta. 2022. Extensible 3D Simulation of Aggregated Systems with FCPP. In *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13271)*, Maurice H. ter Beek and Marjan Sirjani (Eds.). Springer, 55–71. https://doi.org/10.1007/978-3-031-08143-9_4

[12] Giorgio Audrito and Gianluca Torta. 2024. FCPP to aggregate them all. *Sci. Comput. Program.* 231 (2024), 103026. https://doi.org/10.1016/J.SCICO.2023.103026

[13] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. 2013. Organizing the Aggregate: Languages for Spatial Computing. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, Chapter 16, 436–501. https://doi.org/10.4018/978-1-4666-2092-6.ch016

[14] Jacob Beal, Danilo Pianini, and Mirko Viroli. 2015. Aggregate Programming for the Internet of Things. *IEEE Computer* 48, 9 (2015), 22–30. https://doi.org/10.1109/MC.2015.261

[15] Roberto Casadei, Gianluca Aguzzi, Danilo Pianini, and Mirko Viroli. 2023. Programming (and Learning) Self-Adaptive & Self-Organising Behaviour with ScaFi: for Swarms, Edge-Cloud Ecosystems, and More. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2023 - Companion, Toronto, ON, Canada, September 25-29, 2023*. IEEE, 33–34. https://doi.org/10.1109/ACSOS-C58168.2023.00032

[16] Roberto Casadei and Mirko Viroli. 2016. Towards Aggregate Programming in Scala. In *First Workshop on Programming Models and Languages for Distributed Computing* (Rome, Italy) *(PMLDC '16)*. ACM, Article 5, 5:1–5:7 pages. https://doi.org/10.1145/2957319.2957372

[17] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. 2022. ScaFi: A Scala DSL and Toolkit for Aggregate Programming. *SoftwareX* 20 (2022), 101248. https://doi.org/10.1016/J.SOFTX.2022.101248

[18] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. 2014. Specifying and Verifying Properties of Space. In *8th IFIP International Conference in Theoretical Computer Science (TCS) (Lecture Notes in Computer Science, Vol. 8705)*. Springer, 222–235. https://doi.org/10.1007/978-3-662-44602-7_18

[19] Adrian Francalanza, Jorge A. Pérez, and César Sánchez. 2018. Runtime Verification for Decentralised and Distributed Systems. In *Lectures on Runtime Verification - Introductory and Advanced Topics (Lecture Notes in Computer Science, Vol. 10457)*. Springer, 176–210. https://doi.org/10.1007/978-3-319-75632-5_6

[20] Yogi Joshi, Guy Martin Tchamgoue, and Sebastian Fischmeister. 2017. Runtime verification of LTL on lossy traces. In *Proceedings of the Symposium on Applied Computing* (Marrakech, Morocco) *(SAC '17)*. Association for Computing Machinery, New York, NY, USA, 1379–1386. https://doi.org/10.1145/3019612.3019827

[21] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[22] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *J. Log. Algebr. Program.* 78, 5 (2009), 293–303. https://doi.org/10.1016/j.jlap.2008.08.004

[23] Menna Mostafa and Borzoo Bonakdarpour. 2015. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 494–503. https://doi.org/10.1109/IPDPS.2015.95

[24] Laura Nenzi, Luca Bortolussi, Vincenzo Ciancia, Michele Loreti, and Mieke Massink. 2018. Qualitative and Quantitative Monitoring of Spatio-Temporal Properties with SSTL. *Log. Methods Comput. Sci.* 14, 4 (2018). https://doi.org/10.23638/LMCS-14(4:2)2018

[25] Tommy Tracy, Lucas M. Tabajara, Moshe Vardi, and Kevin Skadron. 2020. Runtime Verification on FPGAs with LTLf Specifications. In *2020 Formal Methods in Computer Aided Design (FMCAD)*. 36–46. https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_10

[26] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 2019. From distributed coordination to field calculus and aggregate computing. *J. Log. Algebraic Methods Program.* 109 (2019). https://doi.org/10.1016/j.jlamp.2019.100486

Received 2024-06-24; accepted 2024-07-24