**Optimising Aggregate Monitors for Spatial Logic of Closure Spaces Properties**

# Optimising Aggregate Monitors for Spatial Logic of Closure Spaces Properties

Gianluca Aguzzi
Università di Bologna
Cesena, Italy
gianluca.aguzzi@unibo.it

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

Mirko Viroli
Università di Bologna
Cesena, Italy
mirko.viroli@unibo.it

## Abstract

The advent of highly distributed systems, such as the Internet of Things, has led to the development of distributed systems that require efficient and resilient runtime monitoring. Among the various monitoring techniques, runtime verification is a lightweight verification method that assesses the correctness of a running system concerning a formal specification. In this paper, we investigate the optimization of aggregate monitors, i.e., monitors that operate on ensembles of devices, for properties expressed in Spatial Logic of Closure Spaces (SLCS)—a formal logic designed to reason about spatial relationships between entities in a distributed system. We propose three different algorithms for the implementation of the *somewhere* operator, a key construct in SLCS, and we evaluate their performance through a series of simulations, comparing their convergence time and computational load.

## CCS Concepts

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

## Keywords

runtime verification, aggregate computing, spatial logic

**ACM Reference Format:**
Gianluca Aguzzi, Giorgio Audrito, and Mirko Viroli. 2024. Optimising Aggregate Monitors for Spatial Logic of Closure Spaces Properties. 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, 7 pages. https://doi.org/10.1145/3679008.3685544

## 1 Introduction

Runtime verification (RV) is a lightweight verification technique that assesses the correctness of a running system concerning a formal specification [21]. It complements traditional formal verification methods by providing real-time feedback and monitoring capabilities. While RV has been extensively studied for centralized systems with several well-known tools and techniques [20, 22], its application to distributed settings, such as cyber-physical systems (CPS) and the Internet of Things (IoT), presents unique challenges [16, 31]. These challenges stem from the distributed nature of these systems, including the lack of a global clock, potential communication failures, and the need for decentralized decision-making.

In this paper, we focus on the design of efficient and resilient runtime monitors for distributed systems. Among the current state-of-the-art solutions [24, 26], we leverage the concept of aggregate computing [14], a paradigm that enables the programming of ensembles of devices as a unified computational entity. Aggregate computing provides a high-level abstraction for expressing complex distributed behaviors, making it well-suited for the development of distributed monitors. Specifically, we investigate the optimization of aggregate monitors for properties expressed in Spatial Logic of Closure Spaces (SLCS) [19]. SLCS is a formal logic designed to reason about spatial relationships between entities in a distributed system. By optimizing the evaluation of SLCS properties, we aim to improve the efficiency and responsiveness of runtime monitors in distributed environments. Particularly, the main contribution of this paper consists of the proposal of three different algorithms for the implementation of the *somewhere* operator, a key construct in SLCS. We evaluate the performance of these algorithms through a series of simulations, comparing their convergence time and computational load.

The remainder of this paper is organized as follows. In Sections 2 and 3, we provide background information on distributed runtime verification and aggregate computing. In Section 4, we introduce the Spatial Logic of Closure Spaces (SLCS) and its translation to aggregate computing. Section 5 presents the proposed algorithms for the implementation of the *somewhere* operator. In Section 6, we describe the simulation setup and present the results of our experiments. Finally, Section 7 concludes the paper and outlines future research directions.

## 2 Distributed Runtime Verification

Runtime monitoring is a verification methodology that checks whether a running system adheres to a given specification [25]. This specification can be trace-based or stream-based, where events are mapped as atomic propositions using the logic of the chosen specification language. Examples of such languages include regular expressions and linear temporal logic (LTL) [15]. The extension of this methodology to distributed systems is known as *distributed runtime monitoring*. This significantly increases verification complexity due to the need to manage synchronization, failures, the absence of a global clock, and other issues associated with decentralization and concurrency.


*VORTEX '24, September 19, 2024, Vienna, Austria*

ACM ISBN 979-8-4007-1119-0/24/09
https://doi.org/10.1145/3679008.3685544

In this work, we address the design of runtime monitoring that is both *distributed and decentralized* [21]. Specifically, we assume that each system agent executes its program independently, occasionally synchronizing or communicating with its neighbors using some communication platform. We follow the terminology of Francalanza [21], modeling each agent as a *process*, with each pair of processes considered *remote* from each other. Each process produces a *local trace of events*, described as a sequence of observable values perceived by the agents' sensors or their behavior. The system allows for failures, meaning agents can join or leave the system, so two events in the same trace position do not necessarily occur simultaneously. Monitors must verify system properties by only checking local traces, but are allowed to share internal states with each other periodically. We employ an *online* evaluation, where monitors are executed alongside the local programs (and hosted on the same nodes). For simplicity, we assume each node runs the same monitor. From the perspective of a single monitor, only one trace is managed, even if it contains events received from remote nodes.

Our proposed approach is resilient to failures, a significant challenge in distributed systems. A non-responsive node does not completely disrupt the distributed monitoring process, although it may influence the verdict. Furthermore, our approach advances the state-of-the-art by offering an automatic synthesis of monitors using high-level specifications with logics that handle both spatial and temporal aspects.

## 3 Aggregate Computing

In literature, there is a growing interest in programming ensembles of devices, such as sensor networks, IoT systems, and robotic swarms, as a single computational entity [13, 33]— the so called *macro-programming* paradigm [17]. Among these, *aggregate computing* [14] has emerged as a generalization of previous methods, applicable to distributed networks from the far edge to the fog and cloud [8]. This approach aims to create a programming model that can express complex distributed processes through function composition, grounded in a gossip-like computational process and supporting the reusability of collective adaptive behaviors. Drawing inspiration from "fields" in physics, this model introduces the concept of *computational fields*, which are global data structures that map devices in a distributed system to computational values. These fields can be derived from input fields (such as sensors) either through simple programming constructs at a low level or by composing general-purpose, reusable behavior blocks at a high level, and can ultimately be directed to actuators to create comprehensive adaptive services. Aggregate computing can be understood through an *execution model* (i.e., how effectively the collective computing is performed) and a *programming model* (how it is possible to express collective behaviours).

*Execution model.* In an aggregate computing system, each device performs asynchronous local computations and interacts with its neighbors via local message exchanges. Aggregate computing offers mechanisms to articulate and integrate these distributed computations at a high level of abstraction, eliminating the need for explicit management of message exchanges, device positions, and

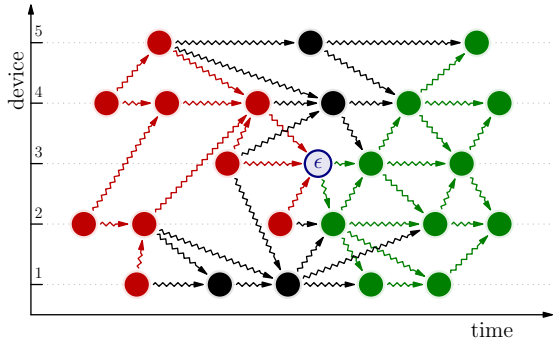



**Figure 1: A sample event structure illustrating different types of event relationships: events $\epsilon'$ (in red) are in the causal past of $\epsilon$ ($\epsilon' < \epsilon$), events $\epsilon''$ (in green) are in the causal future ($\epsilon < \epsilon''$), and concurrent events (non-ordered, in black).**

the node population. A single program executes periodically and asynchronously on each device, following a cyclic schedule:

(1) *(Context acquisition)* the devices collect contextual information from sensors, local memory, and recently received messages, forming a *neighbouring value* that maps neighbor devices $\delta$ to values v;
(2) *(Program evaluation)* during a computation round, the device evaluates the program using the gathered information;
(3) *(Actuations)* the result is stored locally, broadcast to neighbors, and possibly sent to actuators (e.g., motors, robotic arms, user interfaces).

Through repeated execution of the aforementioned rounds across both space (where devices are located) and time (when devices initiate a new cycle), a global behavior emerges [32]. This behavior can be perceived as occurring on the entire network of interconnected devices, modeled as a single *aggregate machine* with a neighboring relation.

*Programming model.* the *eXchange Calculus (XC)* [4], evolving from the *field calculus* [33], is a minimal yet universal language for aggregate computations over networks of mobile devices. XC programs are given a formal semantics via the classical concept of *event structure* [23], which is also used to interpret temporal logic formulas. An *event structure* is defined by a finite set of events $E$ (which can be considered as the evaluation of a program on a device at a given time) and an acyclic *neighbouring relation* $\rightsquigarrow \subseteq E \times E$ representing message passing, namely the transmission of information between devices. Two events $\epsilon$ and $\epsilon'$ are *neighbors* ($\epsilon \rightsquigarrow \epsilon'$) if they are related by the neighboring relation. A sequence of neighbor events $\epsilon_1 \rightsquigarrow \ldots \rightsquigarrow \epsilon_n$ forms a *message path*. Event neighboring induces a *causality relation* $<\subseteq E \times E$, defined as the transitive closure of $\rightsquigarrow$ and modeling causal dependence. Figure 1 illustrates an example structure. There are shown three types of event relationships, casual past (in red), causal future (in green), and concurrent events (in black). An event $\epsilon'$ is in the causal past of another event $\epsilon$ if $\epsilon'$ can influence or is a prerequisite for the occurrence of $\epsilon$. This relationship is denoted as $\epsilon' < \epsilon$, indicating that $\epsilon'$ happened before $\epsilon$ and could causally affect it. Conversely, if events are in a causal future relationship, $\epsilon < \epsilon''$, $\epsilon$ is a prerequisite for $\epsilon''$. Finally, concurrent events are not ordered, meaning that they are not causally related. In practice, event structures emerge

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

**Figure 2: Syntax of SLCS.**

from dynamic device neighborhood graphs. For example, device 3 in Figure 1 initially neighbors devices 4 and 1, but later shifts to neighbors 2 and 4.

We now present small snippets of XC code using standard programming language notation along with two domain-specific functions:

- `nbr(e0, e)`: each device $\delta$ evaluates this expression by broadcasting `e`'s value to neighbors, producing a *neighbouring value*, a *defaulted map* mapping each neighbor $\delta'$ of $\delta$ (including $\delta$ itself) to the latest shared value of `e`, defaulting to $e_0$ if no previous value is available.
- `exchange(e0, (x) => e)`: this expression on each device results from gathering a neighboring value $n$ like `nbr` above, evaluating `e` by substituting $n$ for `x`, and broadcasting the resulting value `v` to neighbors for use in subsequent rounds.

Using this notation, we can express the Bellman-Ford algorithm for distance computation in a network of devices as follows:

```
def dist(source) {
    exchange(infinity, (d) =>
        if (source) {0} else {minHood(d)+1})
}
```

The `dist(source)` function determines the hop-count distance from a source device where the `source` flag is set to true. This function uses an `exchange` construct to propagate distance information across the network. Initially, each device is assigned a distance of infinity ($\infty$). For each device, the `exchange` construct:

(1) Checks if the device is a source (where `source` is true). If so, it assigns a distance of 0.
(2) Otherwise, it calculates the minimum distance among its neighbors (`minHood(d)`) and adds one, indicating an additional hop.

The `minHood(d)` function retrieves the smallest distance from the neighboring field `d`. The algorithm iteratively updates each device's distance based on the information received from its neighbors, eventually converging to the shortest path (in terms of hops) from the source to each device.

## 4 Spatial Logic of Closure Spaces

To represent properties of spatially and temporally distributed systems, spatial logic (SLCS) and past-CTL temporal logic have been studied and can be translated into field calculus monitors [5–7, 11]. Past-CTL logic, akin to CTL + LTL, interprets operators in the past along *message paths* reflecting actual events, allowing runtime computation of truth values within event structures (Fig. 1). This paper focuses on *spatial* logic modalities.

We use $\Diamond$ (closure, true in points "close to" $\phi$) as the primitive modality, defining others as: $\Box\,\phi \triangleq \neg\Diamond\neg\phi$, $\partial\,\phi \triangleq (\Diamond\,\phi) \wedge \neg(\Box\,\phi)$, $\partial^{-}\,\phi \triangleq \phi \wedge \neg(\Box\,\phi)$, $\partial^{+}\,\phi \triangleq (\Diamond\,\phi) \wedge \neg\phi$. The global modalities are:

| | | | |
|---|---|---|---|
| $\top$ | `true` | $q$ | `q()` |
| $\neg\phi$ | `!`$\phi$ | $\phi_1 \vee \phi_2$ | $\phi_1$ `\|\|` $\phi_2$ |
| $\Diamond\,\phi$ | `anyHood(nbr(false,` $\phi$`))` | | |
| $\phi_1\,\mathcal{R}\,\phi_2$ | `if` $(\phi_1)$ `{dist(`$\phi_2$`)<D} else {false}` | | |

**Figure 3: Translation of a primitive set of SLCS operators into XC [5].**

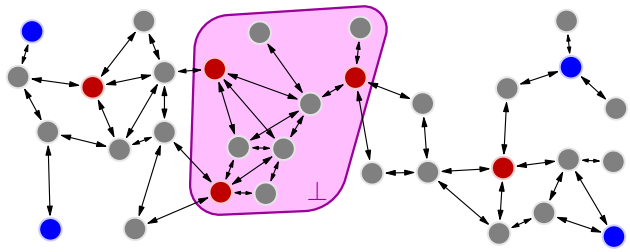
- $\phi\,\mathcal{R}\,\psi$ ($\phi$ reaches $\psi$): true at path endpoints starting at nodes satisfying $\psi$, with $\phi$ holding along the path. Intuitively, along message paths it is possible to reach a node for which $\psi$ holds, while staying within nodes for which $\phi$ holds.
- $\phi\,\mathcal{T}\,\psi$ (touches): true at path endpoints starting at nodes satisfying $\psi$, with $\phi$ holding along the rest of the path. It differs from $\mathcal{R}$ by not requiring the final point where $\psi$ holds to satisfy also $\phi$.
- $\phi\,\mathcal{U}\,\psi$ (surrounded by): true at points within a region where $\phi$ holds, with $\psi$ everywhere on the boundary.
- $\mathcal{G}\,\phi, \mathcal{F}\,\phi$ (everywhere, somewhere): true if $\phi$ holds at every (some) point of every (some) incoming path.

We express other modalities using $\mathcal{R}$ as primitive: $\phi\,\mathcal{T}\,\psi \triangleq \phi\,\mathcal{R}\,\Diamond\,\psi$, $\phi\,\mathcal{U}\,\psi \triangleq \phi \wedge \Box\neg(\neg\psi\,\mathcal{R}\,\neg\phi)$, $\mathcal{F}\,\phi \triangleq \top\,\mathcal{R}\,\phi$, $\mathcal{G}\,\phi \triangleq \neg\mathcal{F}\,\neg\phi$.

*Example 4.1.* As sample application of SLCS, consider the following property to monitor (taken from [5, Ex. 3]): *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.

Figure 3 shows the translation of SLCS formulas into XC, as described in [5], by recursion on sub-formulas. Atomic propositions $q$ translate to `q()`, and logical operators to their XC equivalents. We assume:

- `nbr` and `dist` are as in Section 3.
- `D` is an upper bound on the network diameter.
- `anyHood` collapses Boolean neighbor values into their conjunction.

In the translation of $\phi_1\,\mathcal{R}\,\phi_2$, `dist` is computed within a branch, receiving messages only from neighbors within the

same branch (where $\phi_1$ is true). Thus, it computes the shortest distance from a point satisfying $\phi_2$, restricted to regions where $\phi_1$ holds. Since $\mathcal{F}\,\phi \triangleq \top\,\mathcal{R}\,\phi$, the translation of $\mathcal{R}$ is: `if (`$\phi_1$`) {`$\mathcal{F}(\phi_2)$`} else {false}`. We will therefore focus on optimizing the $\mathcal{F}$ operator, which will inherently optimize the $\mathcal{R}$ operator.

## 5 Translating the Somewhere Operator

Since the SLCS logic is interpreted on (static) graphs, an SLCS monitor cannot exactly compute the value of its formula on the current snapshot of the network in real-time, if this graph can evolve over time. In fact, the XC translation has been shown to be *self-stabilising* to the intended monitor output, that is, it eventually converges to the correct monitor output provided that the network graph does not change for a sufficiently long time. Thus, different translations of the logic could be possible, improving the convergence time and thus the performance in practical scenarios. In this paper, we investigate by simulation four different translations of the $\mathcal{F}$ operator, leading to various trade-offs between convergence speed and computational load. As a first option, we consider the implementation of $\mathcal{F}$ as `dist (`$\phi_2$`)<D` already described in Section 4, as a state-of-the-art baseline. Three more options are described in the following sections.

### 5.1 Knowledge-Free Translation

The baseline implementation relies on a priori knowledge of a (reasonable) upper bound to the network diameter. Underestimating the network diameter prevents the spreading of knowledge to cover the whole network, so it has to be avoided. On the other hand, overestimating the network diameter slows down adjustment when a formula stops being somewhere true. In case such an information is not available, the knowledge-free approach presented in [27] can be used instead. In that paper, a leader election routine is introduced, that computes the minimum among given *identifiers* across a network, under the assumption that identifiers are unique. The algorithm need no further knowledge, as it estimates the network diameter while it is running, and is able to self-adapt to changes in a near-optimal way. In order to implement $\mathcal{F}\,\phi$, we can apply that algorithm using as unique identifiers the pairs $(\neg\phi, \delta)$ where $\delta$ is the identifier of the device. The algorithm will return the lexicographically minimum pair $(b, i)$ in the network, thus privileging true values for $\phi$, and the value of $\mathcal{F}\,\phi$ can then be recovered as $\neg b$.

### 5.2 Replicated-Gossip Translation

A further approach, trading message size for an increase in convergence time, is that given by *replicated gossip* [28]. In this approach, a fixed number $N$ of replicas of a gossip algorithm are kept at all times, each computing whether some device has a true value for $\phi$. These gossip algorithms are very effective in spreading knowledge that the formula is indeed true somewhere, but are unable to spread knowledge that the formula is *not* true somewhere. To address this issue, the replicas are restarted periodically, with an interval of $\Delta$ seconds, with each of the $N$ replicas starting with an offset of $\Delta/N$ seconds from the previous one. At each time, the replicated gossip algorithm will return the value computed by the oldest replica still active. In order for this approach to work, the time $\frac{N-1}{N}\Delta$ needs to be large enough to ensure that knowledge of somewhere $\phi$ being true can propagate to all the network. This time interval effectively determines the convergence time of the algorithm. On the other side, the number of replicas $N$ multiplies the message size exchanged by the algorithm, as each replica contributes equally to the shared data. Similar as in the baseline, it can be upper bounded (increasing the convergence time), but lower-bounding it prevents the results from converging altogether. In order to correctly estimate it, knowledge is needed of multiple parameters: the network diameter, the frequency of communications and the density of devices.

### 5.3 Fastest Translation

Thanks to space-time universality [3], we know that it is always possible to compute in each event any function of its previous events. Thus, we can write a program that has the most recently available knowledge about each device in the network, modelled as a map from device identifiers $\delta$ to pairs $(t, b)$ of timestamps $t$ and Boolean values $b$ computed for formula $\phi$ in device $\delta$ at time $t$. Devices can share those maps between them, always keeping the most recent timestamp for each device. In order to be able to realise that a formula is not somewhere true anymore, when the device realising truth exits the network, older timestamps need to be periodically removed. The interval of time after which a value should be discarded is the same as for replicated-gossip.

## 6 Evaluation

To verify the various implementations of "somewhere" operators, among the many aggregate computing languages available (ScaFi [18], Protelis [30], FCPP [2]) and simulators (Alchemist [29], FCPP [2], ScaFi-Web [1]), we utilized FCPP [2, 12], an aggregate computing language equipped with a simulator [10], since it allows for the easy and efficient implementation of the various algorithms discussed in this paper. The experiments are reproducible and can be found online[1] along with the associated graphs.

*Simulation Setup.* To compare the different operators, we constructed a simulated scenario in which we deploy a group of nodes randomly in a 2D square with a communication radius $c$ of 100 meters. The actual number of deployed nodes and the size of the square depends on two variables: the expected number of hops $h$ along the square diagonal and the density $d$ (average number of neighbours for a device). The number of hops $h$ determines the side $s$ of the square and is calculated as follows:

$$s = \frac{h \times c}{\sqrt{2}}$$

Therefore, the side of the square is proportional to the number of hops. The density $d$, on the other hand, determines the number of nodes within the square. Specifically, the total number of nodes is calculated as follows:

$$n = \frac{d \times s \times s}{\pi \times c \times c}$$

Therefore, the number of nodes is proportional to the square of the side of the square. For instance, given a communication radius of 100 meters, a number of hops $h = 10$, and a density $d = 5$, the side of the square is 707 meters, and the number of nodes is 80.

[1] https://github.com/fcpp-experiments/slcs-optimisation

**Table 1: Simulation free variables.**

| Free variable | Start | End | Step |
|---|---|---|---|
| Hops ($h$) | 1 | 25 | 1 |
| Density ($d$) | 5 | 20 | 2 |
| Seed | 0 | 99 | 1 |
| Speed ($z$) | 0 | 48 | 4 |
| Time interval variance (*tvar*) | 0 | 48 | 2 |
| *Total Simulations*: **7000** | | | |

Additionally, we introduced a randomized movement pattern to the system to assess the robustness of these monitors to changes. During the simulation, nodes select random waypoints and move towards them with a certain predefined speed $z$, selecting a new waypoint when the old one is reached. We express the speed in meters per second, which also corresponds to the percentage of communication radius that can be covered each round, since $c = 100$ and rounds happen about every second.

Each simulation lasts for 300 simulated seconds and executes one of the following algorithms which evaluate the formula $\mathcal{F}(\phi_2)$, labeled as follows: i) *baseline* (`dist(`$\phi_2$`)<D`), ii) *knowledge-free* (Section 5.1), iii) *replicated-gossip* (Section 5.2), and iv) *fastest* (Section 5.3). Each node computes the local program asynchronously with a random period determined according to a Weibull distribution with mean 1 and variance equal to a variable *tvar* (a free variable of the simulation, see Table 1). To test the convergence time of each algorithm, we introduced two events in the system:

- At time = 100 ($t_1$), the property $\phi_2$ is set to true for node 0, which should cause the monitor to evaluate to true across the entire system.
- At time = 200 ($t_2$), the property $\phi_2$ is set back to false for node 0, allowing us to observe the time required for the system to return to a stable state.

Finally, we ran a simulation for each configuration of the free variables and 100 different random seeds (7000 in total) to evaluate the dependence of robustness on network parameters.

*Metrics.* We focus on the following factors to evaluate our system:

- The time required for the system to converge to the correct value.
- The size of the messages exchanged with the neighborhood at each time step (*msiz*).

For the former metric, we compute a local error for each node in the system relative to an ideal value provided by an oracle. An oracle in this case it is just based on verifying the global state in the simulator, therefore having in each time unit the ground truth. Specifically, we consider the true value as 0 if the node is in the correct state and 1 otherwise. Formally, the local error at time $t$ for a node $i$ is defined as:

$$\text{error}(i,t) = \begin{cases} 0 & \text{if oracle}(i,t) = \text{monitor}(i,t) \\ 1 & \text{otherwise} \end{cases}$$

Where monitor is one of the selected algorithm in execution in the whole system. The global error at time $t$ is then computed as the average of the local errors:

$$\text{error}(t) = \frac{1}{n} \sum_{i=1}^{n} \text{error}(i,t)$$

This global error represents the percentage of nodes that are in the incorrect state; hence, a lower value indicates better performance.

*Results.* Figure 4 shows the results of the simulations performed. In general, it can be observed that there is no definitive silver bullet among the various approaches, as none clearly outperforms the others. The only approach that seems to have consistently worse performance is the *knowledge-free* method, as it generally has a higher error compared to the baseline and also consumes more resources. However, it is important to note that, unlike the other methods which require knowledge of the maximum network size, this approach automatically estimates the necessary parameters, making it the best choice in cases of low domain knowledge.

The approach termed *fastest* indeed shows remarkable performance in terms of convergence. As seen in Figure 4a it converges to the correct value in the shortest time, as the error goes to zero almost immediately after the first change. Additionally, the error after the second change (shown Figure 4c, Figure 4d and Figure 4e) does not increase drastically with varying network density, remaining relatively stable. However, this comes at a significant cost in terms of performance, consuming up to $10^4$ bytes for each message exchanged with the neighborhood (w.r.t. $10^1$ for the baseline).

A good compromise appears to be the *replicated* version. Although it takes about 10 times longer to achieve a stable result compared to the *fastest* approach (as shown in the first line), the messages reach a size of only $10^2$ bytes per node (as opposed to $10^4$). Notably, this consumption per node does not increase with the network size but remains constant (as shown in the third line of the graph), since we used a fixed number of 3 replicas.

All approaches seem to be fairly resilient in terms of speed, with *replicated* and *fastest* being the most robust. Naturally, speed does not impact the size of the messages, as the message size is independent of the neighborhood. The *tvar* parameter did not significantly affect the performance of any algorithm, so we do not report the corresponding plots here.

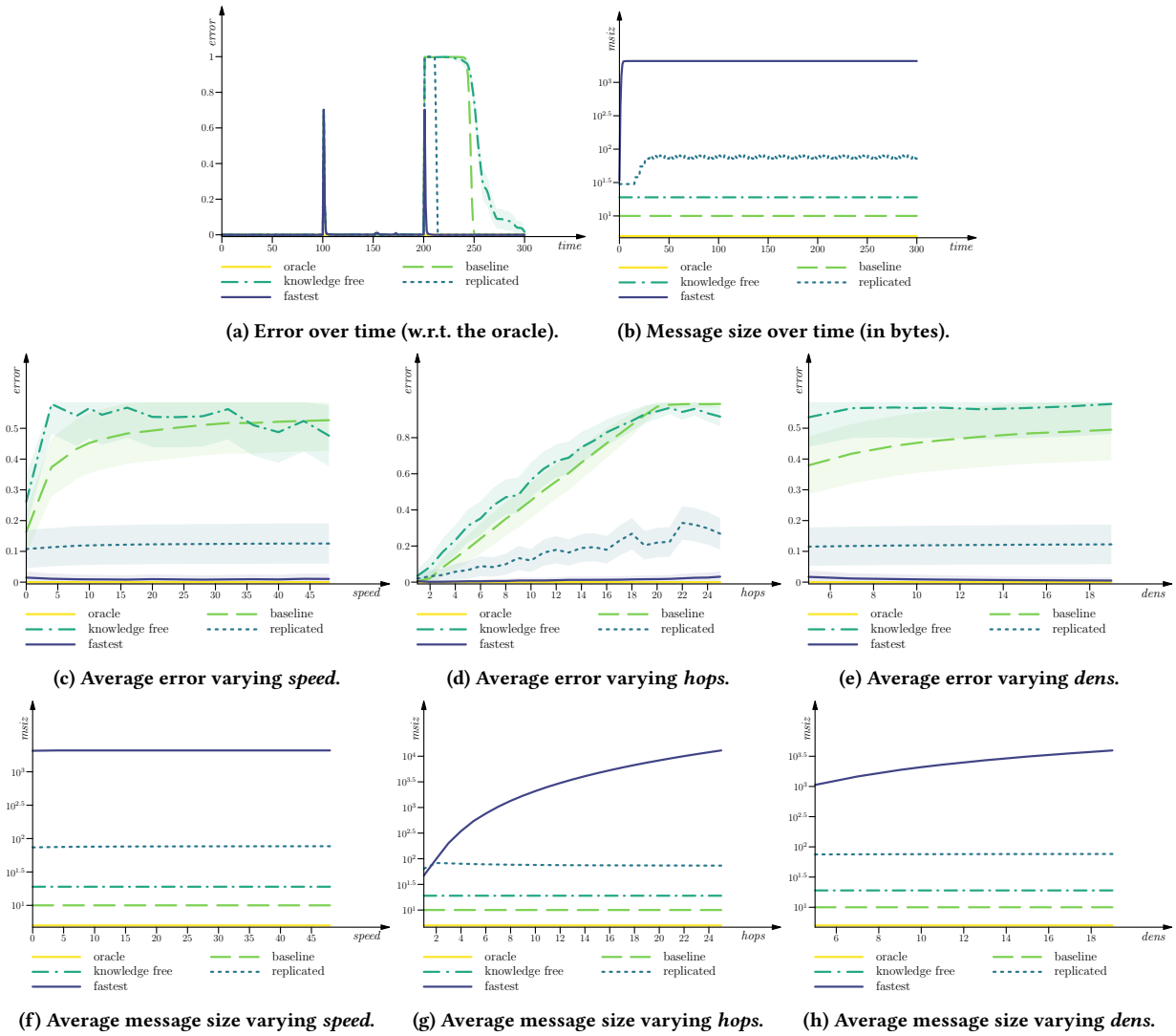
## 7 Conclusion

In this paper, we have explored the problem of efficiently implementing the *somewhere* operator in a distributed runtime verification setting. We have presented three different approaches, each with its own trade-offs between convergence speed and computational load. Our evaluation, based on simulations, has shown that the replicated gossip approach offers a good balance between these two factors, providing fast convergence and a reasonable message size.

Several directions of future work can be pursued from the results in this paper. The experimental analysis could be extended to other spatial operators in other spatial logics other than SLCS. Furthermore, the possibility of combining different approaches to achieve even better performance should be investigated. Finally, theoretical guarantees could be provided to complement experimental results (a preliminary step in that direction has been done in [9]).

## Acknowledgments

This publication has been supported by project NODES, funded by the MUR – M4C2 1.5 of PNRR with grant agreement no. ECS00000036, and by the Italian PRIN project "CommonWears" (2020HCWWLP).




(a) Error over time (w.r.t. the oracle).

(b) Message size over time (in bytes).

(c) Average error varying *speed*.

(d) Average error varying *hops*.

(e) Average error varying *dens*.

(f) Average message size varying *speed*.

(g) Average message size varying *hops*.

(h) Average message size varying *dens*.

**Figure 4: Results of the comparison of different versions of somewhere. Each line represents the average of the metric over all the simulations, with the shaded area representing the standard deviation. The first line shows the results over time in the case with ($d = 10$, $h = 10$, $tvar = 10$, $s = 10$). The second line shows the average error varying various simulation parameters. The last line shows the average message size varying various simulation parameters.**

## References


[1] Gianluca Aguzzi, Roberto Casadei, Niccolò Maltoni, Danilo Pianini, and Mirko Viroli. 2021. ScaFi-Web: A Web-Based Application for Field-Based Coordination Programming. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 285–299. https://doi.org/10.1007/978-3-030-78142-2_18

[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. 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

[5] 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

[6] Giorgio Audrito, Ferruccio Damiani, Giuseppe Martino Di Giuda, Silvia Meschini, Laura Pellegrini, Elena Seghezzi, Lavinia Chiara Tagliabue, Lorenzo Testa, and Gianluca Torta. 2021. RM for users' safety and security in the built environment. In *VORTEX 2021: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution, Virtual Event, Denmark, 12 July 2021*, Wolfgang Ahrendt, Davide Ancona, and Adrian Francalanza (Eds.). ACM, 13–16. https://doi.org/10.1145/3464974.3468445

[7] 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

[8] 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

[9] Giorgio Audrito, Ferruccio Damiani, and Gianluca Torta. 2024. Real-Time Guarantees for SLCS Monitors in XC. In *VORTEX 2024: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution*. ACM. https://doi.org/10.1145/3679008.3685545

[10] 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

[11] Giorgio Audrito and Gianluca Torta. 2021. Towards aggregate monitoring of spatio-temporal properties. In *VORTEX 2021: Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution, Virtual Event, Denmark, 12 July 2021*, Wolfgang Ahrendt, Davide Ancona, and Adrian Francalanza (Eds.). ACM, 26–29. https://doi.org/10.1145/3464974.3468448

[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] Pierfrancesco Bellini, R. Mattonlini, and Paolo Nesi. 2000. Temporal logics for real-time system specification. *ACM Comput. Surv.* 32, 1 (2000), 12–42. https://doi.org/10.1145/349194.349197

[16] Marc Carwehl, Thomas Vogel, Genaína Nunes Rodrigues, and Lars Grunske. 2024. Runtime Verification of Self-Adaptive Systems with Changing Requirements. In *Software Engineering 2024, Fachtagung des GI-Fachbereichs Softwaretechnik, Linz, Austria, February 26 - March 1, 2024 (LNI, Vol. P-343)*, Rick Rabiser, Manuel Wimmer, Iris Groher, Andreas Wortmann, and Bianca Wiesmayr (Eds.). Gesellschaft für Informatik e.V., 151–152. https://doi.org/10.18420/SW2024_50

[17] Roberto Casadei. 2023. Macroprogramming: Concepts, State of the Art, and Opportunities of Macroscopic Behaviour Modelling. *ACM Comput. Surv.* 55, 13s (2023), 275:1–275:37. https://doi.org/10.1145/3579353

[18] 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

[19] Vincenzo Ciancia, Diego Latella, Michele Loreti, and Mieke Massink. 2016. Model Checking Spatial Logics for Closure Spaces. *Log. Methods Comput. Sci.* 12, 4 (2016). https://doi.org/10.2168/LMCS-12(4:2)2016

[20] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* 23, 2 (2021), 255–284. https://doi.org/10.1007/S10009-021-00609-Z

[21] 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

[22] Klaus Havelund and Grigore Rosu. 2004. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods Syst. Des.* 24, 2 (2004), 189–215. https://doi.org/10.1023/B:FORM.0000017721.39909.4B

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

[24] K. Rustan M. Leino. 2013. Developing verified programs with dafny. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 1488–1490. https://doi.org/10.1109/ICSE.2013.6606754

[25] 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

[26] Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12225)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 190–202. https://doi.org/10.1007/978-3-030-53291-8_12

[27] Yuanqiu Mo, Giorgio Audrito, Soura Dasgupta, and Jacob Beal. 2022. Near-optimal knowledge-free resilient leader election. *Autom.* 146 (2022), 110583. https://doi.org/10.1016/J.AUTOMATICA.2022.110583

[28] Danilo Pianini, Jacob Beal, and Mirko Viroli. 2016. Improving Gossip Dynamics Through Overlapping Replicates. In *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9686)*, Alberto Lluch-Lafuente and José Proença (Eds.). Springer, 192–207. https://doi.org/10.1007/978-3-319-39519-7_12

[29] Danilo Pianini, Sara Montagna, and Mirko Viroli. 2013. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation* 7, 3 (2013), 202–215. https://doi.org/10.1057/jos.2012.27

[30] Danilo Pianini, Mirko Viroli, and Jacob Beal. 2015. Protelis: Practical Aggregate Programming. In *ACM Symposium on Applied Computing (SAC)*. 1846–1853. https://doi.org/10.1145/2695664.2695913

[31] César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. 2019. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.* 54, 3 (2019), 279–335. https://doi.org/10.1007/S10703-019-00337-W

[32] Mirko Viroli, Giorgio Audrito, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. 2018. Engineering Resilient Collective Adaptive Systems by Self-Stabilisation. *ACM Trans. Model. Comput. Simul.* 28, 2 (2018), 1–16.

[33] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 2018. From Field-Based Coordination to Aggregate Computing. In *Coordination Models and Languages (COORDINATION) (Lecture Notes in Computer Science, Vol. 10852)*. Springer, 252–279. https://doi.org/10.1007/978-3-319-92408-3_12


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