Decentralized Predicate Detection over Partially Synchronous Continuous-Time Signals*

Charles Koll[1], Anik Momtaz[2], Borzoo Bonakdarpour[2], and Houssam Abbas[1,3]

[1] Oregon State University, USA {kollch,houssam.abbas}@oregonstate.edu
[2] Michigan State University, USA {momtazan,borzoo}@msu.edu
[3] Corresponding author.

Abstract. We present the first decentralized algorithm for detecting predicates over continuous-time signals under partial synchrony. A distributed cyber-physical system (CPS) consists of a network of agents, each of which measures (or computes) a continuous-time signal. Examples include distributed industrial controllers connected over wireless networks and connected vehicles in traffic. The safety requirements of such CPS, expressed as logical predicates, must be monitored at runtime. This monitoring faces three challenges: first, every agent only knows its own signal, whereas the safety requirement is global and carries over multiple signals. Second, the agents' local clocks drift from each other, so they do not even agree on the time. Thus, it is not clear which signal values are actually synchronous to evaluate the safety predicate. Third, CPS signals are continuous-time so there are potentially uncountably many safety violations to be reported. In this paper, we present the first decentralized algorithm for detecting conjunctive predicates in this setup. Our algorithm returns all possible violations of the predicate, which is important for eliminating bugs from distributed systems regardless of actual clock drift. We prove that this detection algorithm is in the same complexity class as the detector for discrete systems. We implement our detector and validate it experimentally.

Keywords: Predicate detection · Distributed systems · Partial synchrony · Cyber-physical systems.

1 Introduction: Detecting All Errors in Distributed CPS

This paper studies the problem of detecting all property violations in a distributed cyber-physical systems (CPS). A *distributed CPS* consists of a network of communicating agents. Together, the agents must accomplish a common task and preserve certain properties. For example, a network of actuators in an industrial control system must maintain a set point, or a swarm of drones must maintain a certain geometric formation. In these examples, we have N agents generating N continuous-time and real-valued signals $x_n, 1 \leq n \leq N$, and a

* Supported by NSF SHF awards 2118179 and 2118356.

global property of all these signals must be maintained, such as the property $(x_1 > 0) \wedge \ldots (x_N > 0)$. At runtime, an algorithm continuously monitors whether the property holds.

These systems share the following characteristics: first, CPS signals are *analog* (continuous-time, real-valued), and the global properties are continuous-time properties. From a distributed computing perspective, this means that every moment in continuous-time is an event, yielding uncountably many events. Existing reasoning techniques from the discrete time settings, by contrast, depend on there being at the most countably many events.

Second, each agent in these CPS has a *local clock* that drifts from other agents' clocks: so if agent 1 reports $x_1(3) = 5$ and agent 2 reports that $x_2(3) = -10$, these are actually not necessarily synchronous measurements. So we must redefine property satisfaction to account for unknown drift between clocks. For example, if the local clocks drift by at most 1 second, then the monitor must actually check whether any value combination of $x_1(t), x_2(s)$ violates the global property, with $|t - s| \leq 1$. We want to identify any scenario where the system execution *possibly* [5] violates the global property; the actual unknown execution may or may not do so.

Clock drift raises a third issue: the designers of distributed systems want to know *all the ways* in which an error state could occur. E.g., suppose again that the clock drift is at most 1, and the designer observes that the values $(x_1(1), x_2(1.1))$ violate the specification, and eliminates this bug. But when she reruns the system, the actual drift is 0.15 and the values $(x_1(1), x_2(1.15))$ also violate the spec. Therefore all errors, resulting from all possible clock drifts within the bound, must be returned to the designers. This way the designers can guarantee the absence of failures regardless of the actual drift amount. When the error state is captured in a predicate, this means that all possible satisfactions of the predicate must be returned. This is known as the *predicate detection* problem. We distinguish it from predicate monitoring, which requires finding only one such satisfaction, not all.

Finally, in a distributed system, a central monitor which receives all signals is a single point of failure: if the monitor fails, predicate detection fails. Therefore, ideally, the detection would happen in decentralized fashion.

In this work, we solve the problem of decentralized predicate detection for distributed CPS with drifting clocks under partial synchrony.

Related Work There is a rich literature dealing with decentralized predicate detection in *the discrete-time setting*: e.g. [4] detects regular discrete-time predicates, while [7] detects lattice-linear predicates over discrete states, and [18] performs detection on a regular subset of Computation Tree Logic (CTL). We refer the reader to the books by Garg [6] and Singhal [10] for more references. By contrast, we are concerned with *continuous-time* signals, which have uncountably many events and necessitate new techniques. For instance, one cannot directly iterate through events as done in the discrete setting.

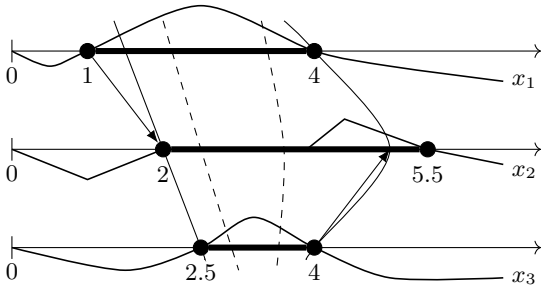


Fig. 1: An example of a continuous-time distributed signal with 3 agents. Three timelines are shown, one per agent. The signals x_n are also shown, and the local time intervals over which they are non-negative are solid black. The skew ϵ is 1. The Happened-before relation is illustrated with solid arrows, e.g. between $e_1^1 \to e_2^2$, and $e_3^4 \to e_2^5$. These are not message transmission events, rather they follow from Definition 3. Some satisfying cuts for the predicate $\phi = (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_3 \geq 0)$ are shown as dashed arcs, and the extremal cuts as solid arcs. All extremal cuts contain root events, and leftmost cut A also contains non-root events.

The recent works [15,14] do *monitoring* of temporal formulas over partially synchronous analog distributed systems - i.e., they only find one satisfaction, not all. Moreover, their solution is centralized.

More generally, one finds much work on monitoring temporal logic properties, especially Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL), but they either do monitoring or work in discrete-time. Notably, [1] used a three-valued MTL for monitoring in the presence of failures and non-FIFO communication channels. [3] monitors satisfaction of an LTL formula. [16] considered a three-valued LTL for distributed systems with asynchronous properties. [2] addressed the problem with a tableau technique for three-valued LTL. Finally, [19] considered a past-time distributed temporal logic which emphasizes distributed properties over time.

Illustrative Example. It is helpful to overview our algorithm and key notions via an example before delving into the technical details. An example is shown in Figure 1. Three agents produce three signals x_1, x_2, x_3. The decentralized detector consists of three local detectors D_1, D_2, D_3, one on each agent. Each x_n is observed by the corresponding D_n. The predicate $\phi = (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_3 \geq 0)$ is being detected. It is possibly true over the intervals shown with solid black bars; their endpoints are measured on the local clocks. The detector only knows that the maximal clock skew is $\epsilon = 1$, but not the actual value, which might be time-varying.

Because of clock skew, any two local times within ϵ of each other must be considered as potentially concurrent, i.e. they might be measured at a truly synchronous moment. For example, the triple of local times $[4, 4.5, 3.6]$ might

have been measured at the global time 4, in which case the true skews were 0, 0.5, and -0.4 respectively. Such a triple is (loosely speaking) called a *consistent cut* (Definition 4). The detector's task is to find all consistent cuts that satisfy the predicate. In continuous time, there can be uncountably many, as in Figure 1; the dashed lines show two satisfying consistent cuts, or satcuts for short.

In this example, our detector outputs two satcuts, $[1.5, 2, 2.5]$ and $[4, 5, 4]$, shown as thin solid lines. These two have the special property (shown in this paper) that every satcut lies between them, and every cut between them is a satcut. For this reason we call them *extremal satcuts* (Definition 6). Thus these two satcuts are a finite representation of the uncountable set of satcuts, and encode all the ways in which the predicate might be satisfied.

We note three further things: the extremal satcuts are not just the endpoints of the intervals, and simply inflating each interval by ϵ and intersecting them does not yield the satcuts. Each local detector must somehow learn of the relevant events (and only those) on other agents, to determine whether they constitute extremal satcuts.

Contributions In this paper, we present the first *decentralized predicate detector for distributed CPS*, thus enhancing the rigor of distributed CPS design.

- Our solution is fully decentralized: each agent only ever accesses its own signal, and exchanges a limited amount of information with the other agents.
- It is an online algorithm, running simultaneously with the agents' tasks.
- It applies to an important class of global properties that are conjunctions of local propositions.
- We introduce a new notion of clock, the *physical vector clock*, which might be of independent interest. A physical vector clock orders continuous-time events in a distributed computation without a shared clock.
- Our algorithm can be deployed on top of existing infrastructure. Specifically, our algorithm includes a modified version of the classical detector of [4], and so can be deployed on top of existing infrastructure which already supports that detector.

Organization In Section 2, we give necessary definitions and define the problem. In Section 3 we establish fundamental properties of the uncountable set of events S_E satisfying the predicate. Our detector is made of two processes: a decentralized abstractor presented in Section 4, and a decentralized slicer presented in Section 5. Together, they compute a finite representation of the uncountable S_E. The complexity of the algorithm is also analyzed in Section 5. Section 6 demonstrates an implementation of the detector, and Section 7 concludes. All proofs are in the report [9].

2 Preliminaries and Problem Definition

We first set some notation. The set of reals is $\mathbb{R}$, the set of non-negative reals is $\mathbb{R}_{\geq 0}$. The integer set $\{1, \ldots, N\}$ is abbreviated as $[N]$. *Global* time values (kept

by an *imaginary* global clock) are denoted by χ, χ', χ_1, χ_2, etc, while the symbols $t, t', t_1, t_2, s, s', s_1, s_2$, etc. denote *local* clock values specific to given agents which will always be clear from the context. A *lattice* is a set S equipped with a partial order relation $\sqsubseteq$ s.t. every 2 elements have a supremum, or *join*, and an infimum, or *meet*. An *increasing* function f is one s.t. $t < t' \implies f(t) < f(t')$. Notation $(x_n)_n$ indicates a sequence $(x_1, \ldots, x_N)$ where N is always clear from context.

2.1 The Continuous-Time Setup

This section defines the setup of this study. It generalizes the classical discrete-time setup, and follows closely the setup in [15]. We assume a loosely coupled system with asynchronous message passing between agent monitors. Specifically, the system consists of N reliable *agents* that do not fail, denoted by $\{A_1, A_2, \ldots, A_N\}$, without any shared memory or global clock. The output signal of agent A_n is denoted by x_n, for $1 \leq n \leq N$. Agents can communicate via FIFO lossless channels. There are no bounds on message transmission times.

In the discrete-time setting, an event is a value change in an agent's variables. The following definition generalizes this to the continuous-time setting.

Definition 1 (Output signal and events). *An* output signal *(of some agent) is a function $x : \mathbb{R}_{\geq 0} \to \mathbb{R}$, which is right-continuous (i.e., $\lim_{s \to t^+} x(s) = x(t)$ at every t), and left-limited (i.e., $\lim_{s \to t^-} |x(s)| < \infty$ for all t).*

In an agent A_n, an event *is a pair $(t, x_n(t))$, where t is the local time kept by the agent's local clock. This will often be abbreviated as e_n^t to follow standard notation from the discrete-time literature.*

Note that an output signal can contain discontinuities.

Definition 2 (Left and right roots). *A* root *is an event e_n^t where $x_n(t) = 0$ or a discontinuity at which the signal changes sign: $sgn(x_n(t)) \neq sgn(\lim_{s \to t^-} x_n(s))$. A* left root *$e_n^t$ is a root preceded by negative values: there exists a positive real δ s.t. $x_n(t-\alpha) < 0$ for all $0 < \alpha \leq \delta$. A* right root *e_n^t is a root followed by negative values: $x_n(t+\alpha) < 0$ for all $0 < \alpha \leq \delta$.*

In Figure 1, the only left root of x_2 is $e_2^2 = (2, x_2(2)) = (2, 0)$. The single right root of x_2 is $e_2^{5.5}$. Notice that intervals where the signal is identically 0 are allowed, as in x_2.

We will need to refer to a global clock which acts as a 'real' time-keeper. This global clock is a theoretical object used in definitions and theorems, and is *not* available to the agents. We make these assumptions:

Assumption 1. *(a) (Partial synchrony) The local clock of an agent A_n is an increasing function $c_n : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_n(\chi)$ is the value of the local clock at global time χ. For any two agents A_n and A_m, we have:*

$$\forall \chi \in \mathbb{R}_{\geq 0} : |c_n(\chi) - c_m(\chi)| < \epsilon$$

with $\epsilon > 0$ being the maximum clock skew. *The value ϵ is known by the detector in the rest of this paper. In the sequel, we make it explicit when we refer to 'local' or 'global' time.*

(b) (Starvation-freedom and non-Zeno) Every signal x_n has infinitely many roots in $\mathbb{R}_{\geq 0}$, with a finite number of them occurring in any bounded interval.

Remark 1. Our detection algorithm can trivially handle multi-dimensional output signals x_n. We skip this generalization for clarity of exposition.

Remark 2. In distributed systems, agents typically exchange messages as part of normal operation. These messages help establish an ordering between events (a Send occurs before the corresponding Receive). This extra order information can be incorporated in our detection algorithm with extra bookkeeping.
We do *not* assume that the clock drift is constant - it can vary with time. It is assumed to be uniformly bounded by ϵ, which can be achieved by using a clock synchronization algorithm, like NTP [13].

A distributed signal is modeled as a set of events partially ordered by Lamport's *happened-before* relation [11], adapted to the continuous-time setting.

Definition 3 (Analog Distributed signal). *A* distributed signal *on N agents is a tuple $(E, \rightarrow)$, in which E is a set of events*

$$E = \{e_n^t \mid n \in [N],\ t \in \mathbb{R}_{\geq 0}\}$$

such that for all $t \in \mathbb{R}_{\geq 0}, n \in [N]$, there exists an event e_n^t in E, and t is local time in agent A_n. The happened-before *relation $\rightarrow \subseteq E \times E$ between events is such that:*

(1) In every agent A_n, all events are totally ordered, that is,

$$\forall t, t' \in \mathbb{R}_{\geq 0} : (t < t') \implies (e_n^t \rightarrow e_n^{t'}).$$

(2) For any two events $e_n^t, e_m^{t'} \in E$, if $t + \epsilon \leq t'$, then $e_n^t \rightarrow e_m^{t'}$.
(3) If $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

We denote $E[n]$ the subset of events that occur on A_n, i.e. $E[n] := \{e_n^t \in E\}$

The happened-before relation, $\rightarrow$, captures what can be known about event ordering in the absence of perfect synchrony. Namely, events on the same agent can be linearly ordered, and at least an ϵ of time must elapse between events on different agents for us to say that one happened before the other. Events from different agents closer than an ϵ apart are said to be *concurrent*.

Conjunctive Predicates This paper focuses on specifications expressible as *conjunctive predicates* ϕ, which are conjunctions of N linear inequalities.

$$\phi := (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge \ldots \wedge (x_N \geq 0). \tag{1}$$

These predicates model the simultaneous co-occurrence, in global time, of events of interest, like 'all drones are dangerously close to each other'. Eq. (1) also captures the cases where some conjuncts are of the form $x_n \leq 0$ and $x_n = 0$. If

N numbers (a_n) satisfy predicate ϕ (i.e., are all non-negative), we write this as $(a_1, \ldots, a_N) \models \phi$. Henceforth, we say 'predicate' to mean a conjunctive predicate. Note that the restriction to linear inequalities does not significantly limit our ability to model specifications. If an agent n has some signal x_n with which we want to check $f(x_n) \geq 0$ for some arbitrary function f, then the agent can generate an auxiliary signal $y_n := f(x_n)$ so that we can consider the linear inequality $y_n \geq 0$.

What does it mean to say that a distributed signal satisfies ϕ? And at what moment in time? In the ideal case of perfect synchrony ($\epsilon = 0$) we'd simply say that E satisfies ϕ at χ whenever $(x_1(\chi), \ldots, x_N(\chi)) \models \phi$. We call such a synchronous tuple $(x_n(\chi))_n$ a *global state*. But because the agents are only synchronized to within an $\epsilon > 0$, it is not possible to guarantee evaluation of the predicate at true global states. The conservative thing is to treat concurrent events, whose local times differ by less than ϵ, as being simultaneous on the global clock. E.g., if $N = 2$ and $\epsilon = 1$ then $(x_1(1), x_2(1.5))$ is treated as a possible global state. The notion of consistent cut, adopted from discrete-time distributed systems [8], formalizes this intuition.

Definition 4 (Consistent Cut). *Given a distributed signal $(E, \to)$, a subset of events $C \subset E$ is said to form a* consistent cut *if and only if when C contains an event e, then it contains all events that happened-before e. Formally,*

$$\forall e \in E : (e \in C) \wedge (f \to e) \implies f \in C. \tag{2}$$

We write $C[n]$ for the cut's local events produced on A_n, and $C^\tau[n] := \{t \mid e_n^t \in C[n]\}$ for the timestamps of a cut's local events.

From this and Definition 3 (3) it follows that if $e_m^{t'}$ is in C, then C also contains every event e_n^t such that $t + \epsilon \leq t'$. Thus to avoid trivialities, we may assume that C contains at least one event from every agent.

A consistent cut C is represented by its *frontier* $\mathsf{front}(C) = \left(e_1^{t_1}, \ldots, e_N^{t_N}\right)$, in which each $e_n^{t_n}$ is the last event of agent A_n appearing in C. Formally:

$$\forall n \in [N],\ t_n := \sup C^\tau[n] = \sup\{t \in \mathbb{R}_{\geq 0} \mid\ e_n^t \in C[n]\}.$$

Henceforth, we simply say 'cut' to mean a consistent cut, and we denote a frontier by $(e_n^{t_n})_n$. We highlight some easy yet important consequences of the definition: on a given agent A_n, $e_n^t \in C$ for all $t < t_n$, so the timestamps of the cut's local events, $C^\tau[n]$, form a left-closed interval of the form $[0, a]$, $[0, a)$ or $[0, \infty)$. Moreover, either $C^\tau[n] = [0, \infty)$ for all n, in which case $C = E$, or every $C^\tau[n]$ is bounded, in which case every t_n is finite and $|t_n - t_m| \leq \epsilon$ for all n, m. *Thus the frontier of a cut is a possible global state.* This then justifies the following definition of distributed satisfaction.

Definition 5 (Distributed Satisfaction; S_E). *Given a predicate ϕ, a distributed signal $(E, \to)$ over N agents, and a consistent cut C of E with frontier*

$$\mathsf{front}(C) = \Big((t_1, x_1(t_1)), \ldots, (t_N, x_N(t_N)) \Big)$$

we say that C satisfies ϕ *iff* $\big(x_1(t_1), x_2(t_2), \ldots, x_N(t_N)\big) \models \phi$. *We write this as* $C \models \phi$, *and say that* C *is a* satcut. *The set of all satcuts in* E *is written* S_E.

2.2 Problem Definition: Decentralized Predicate Detection

The detector seeks to find *all* possible global states that satisfy a given predicate, i.e. all satcuts in S_E. In general, S_E is uncountable.

Architecture. The system consists of N agents with partially synchronous clocks with drift bounded by a known ϵ, generating a continuous-time distributed signal $(E, \rightarrow)$. Agents communicate in a FIFO manner, where messages sent from an agent A_1 to an agent A_2 are received in the order that they were sent.

Problem statement. Given $(E, \rightarrow)$ and a conjunctive predicate ϕ, find a decentralized detection algorithm that computes a finite representation of S_E. The detector is decentralized, meaning that it consists of N local detectors, one on each agent, with access only to the local signal x_n (measured against the local clock), and to messages received from other agents' detectors.

By computing a representation of all of S_E (and not some subset), we account for asynchrony and the unknown orderings of events within ϵ of each other. One might be tempted to propose something like the following algorithm: detect all roots on all agents, then see if any N of them are within ϵ of each other. This quickly runs into difficulties: first, a satisfying cut is not necessarily made up of roots; some or all of its events can be interior to the intervals where x_n's are positive (see Figure 2). Second, the relation between roots and satcuts must be established: it is not clear, for example, whether even satcuts made of only roots are enough to characterize all satcuts (it turns out, they're not). Third, we must carefully control how much information is shared between agents, to avoid the detector degenerating into a centralized solution where everyone shares everything with everyone else.

3 The Structure of Satisfying Cuts

We establish fundamental properties of satcuts. In the rest of this paper we exclude the trivial case $C = E$. Proposition 1 mirrors a discrete-time result [4].

Proposition 1. *The set of satcuts for a conjunctive predicate is a lattice where the join and meet are the union and intersection operations, respectively.*

We show that the set of satcuts is characterized by special elements, which we call the leftmost and rightmost cuts.

Definition 6 (Extremal cuts). *Let* S_E *be the* set *of all satcuts in a given distributed signal* $(E, \rightarrow)$. *For an arbitrary* $C \in S_E$ *with frontier* $(e_n^{t_n})_n$ *and positive real* α, *define* $C - \alpha$ *to be the set of cuts whose frontiers are given by*

$$(e_1^{t_1-\delta_1}, e_2^{t_2-\delta_2}, \ldots, e_N^{t_N-\delta_N}) \text{ s.t. for all } n : 0 \leq \delta_n \leq \alpha \text{ and for some } n.\ \delta_n > 0$$

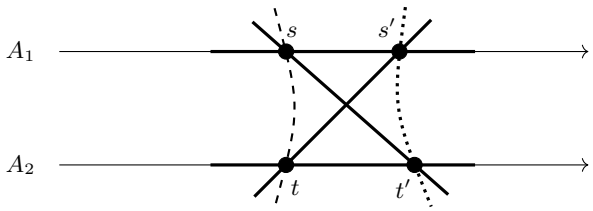


Fig. 2: Two satcuts for a pair of agents A_1 and A_2, shown by the solid lines (s, t') and (s', t). Their intersection is (s, t), shown by a dashed arc, and their union is (s', t'), shown by a dotted arc. For a conjunctive predicate ϕ, the intersection and union are also satcuts, forming a lattice of satcuts.

A leftmost satcut *is a satcut $C \in S_E$ for which there exists a positive real α s.t. $C - \alpha$ and S_E do not intersect. The set $C + \alpha$ is similarly defined. A* rightmost cut *C (not necessarily sat) is one for which there exists a positive real α s.t. $C + \alpha$ and S_E do not intersect, and $C - \alpha \subset S_E$. We refer to leftmost and rightmost (sat)cuts as* extremal cuts*.*

Intuitively, $C - \alpha$ ($C + \alpha$) is the set of all cuts one obtains by slightly moving the frontier of C to the left (right) by amounts less than α. If doing so always yields non-satisfying cuts, then C is a leftmost satcut. If moving C slightly to the right always yields unsatisfying cuts, but moving it slightly left yields satcuts, then C is a rightmost cut. The reason we don't speak of rightmost *sat*cuts is that we only require signals to be left-limited, not continuous. If signals x_n are all continuous, then rightmost cuts are all satisfying as well.

In a signal, there are multiple extremal cuts. Figure 2 suggests, and Lemma 1 proves, that all satcuts live between a leftmost satcut and rightmost cut.

Lemma 1 (Satcut intervals). *Every satcut of a conjunctive predicate lies in-between a leftmost satcut and rightmost cut, and there are no non-satisfying cuts between a leftmost satcut and the first rightmost cut that is greater than it in the lattice order.*

Thus we may visualize satcuts as forming N-dimensional intervals with endpoints given by the extremal cuts. The main result of this section states that there are finitely many extremal satcuts in any bounded time interval, so the extremal satcuts are the finite representation we seek for S_E.

Theorem 1. *A distributed signal has finitely many extremal satcuts in any bounded time interval.*

Therefore, it is conceivably possible to recover algorithmically the extremal cuts, and therefore all satcuts by Lemma 1. The rest of this paper shows how.

4 The Abstractor Process

Having captured the structure of satcuts, we now define the distributed *abstractor process* that will turn our continuous-time problem into a discrete-time one,

amenable to further processing by our modified version of the slicer algorithm of [4]. This abstractor also has the task of creating a happened-before relation. We first note a few complicating factors. First, this will not simply be a matter of sampling the roots of each signal. That is because extremal cuts can contain non-root events, as shown in Figure 1. Thus the abstractor must somehow find and sample these non-root events as part of its operation. Second, as in the discrete case, we need a kind of clock that allows the local detector to know the happened-before relation between events. The local timestamp of an event, and existing clock notions, are not adequate for this. Third, to establish the happened-before relation, there is a need to exchange event information between the processes, without degenerating everything into a centralized process (by sharing everything with everyone). This complicates the operation of the local abstractors, but allows us to cut the number of messages in half.

4.1 Physical Vector Clocks

We first define Physical Vector Clocks (PVCs), which generalize vector clocks [12] from countable to uncountable sets of events. They are used by the abstractor process (next section) to track the happened-before relation. A PVC captures one agent's knowledge, at appropriate local times, of events at other agents.

Definition 7 (Physical Vector Clock). *Given a distributed signal $(E, \rightarrow)$ on N agents, a* Physical Vector Clock*, or PVC, is a set of N-dimensional timestamp vectors $\mathbf{v}_n^t \in \mathbb{R}^N_{\geq 0}$, where vector $\mathbf{v}_n^t$ is defined by the following:*

(1) Initialization: $\mathbf{v}_n^0[i] = 0, \quad \forall i \in \{1, \ldots, N\}$
(2) Timestamps store the local time of their agent: $\mathbf{v}_n^t[n] = t$ *for all* $t > 0$.
(3) Timestamps keep a consistent view of time: Let V_n^t *be the set of all timestamps* $\mathbf{v}_m^s$ *s.t.* e_m^s *happened-before* e_n^t *in* E*. Then:*

$$\mathbf{v}_n^t[i] = \max_{\mathbf{v}_m^s \in V_n^t} (\mathbf{v}_m^s[i]), \quad \forall i \in [N] \setminus \{n\}, t > 0$$

PVCs are partially ordered: $\mathbf{v}_n^t < \mathbf{v}_m^{t'}$ *iff* $\mathbf{v}_n^t \neq \mathbf{v}_m^{t'}$ *and* $\mathbf{v}_n^t[i] \leq \mathbf{v}_m^{t'}[i]$ $\forall i \in [N]$.

We say $\mathbf{v}_n^t$ is *assigned* to e_n^t. The detection algorithm can now know the happened-before relation by comparing PVCs.

Theorem 2. *Given a distributed signal $(E, \rightarrow)$, let V be the corresponding set of PVC timestamps. Then $(V, <)$ and $(E, \rightarrow)$ are order isomorphic, i.e., there is a bijective mapping between V and E s.t. $e_n^t \rightarrow e_m^{t'}$ iff $\mathbf{v}_n^t < \mathbf{v}_m^{t'}$.*

Definition 7 is not quite a constructive definition. We need a way to actually compute PVCs. This is enabled by the next theorem.

Theorem 3. *The assignment*

$$\mathbf{v}_n^t = \begin{cases} [0, \ldots, 0, t, 0, \ldots, 0], & t < \epsilon \\ [t - \epsilon, \ldots, t - \epsilon, t, t - \epsilon, \ldots, t - \epsilon], & t \geq \epsilon \end{cases}$$

where the t is in the n^{th} position in both cases, satisfies the conditions of PVC in Definition 7.

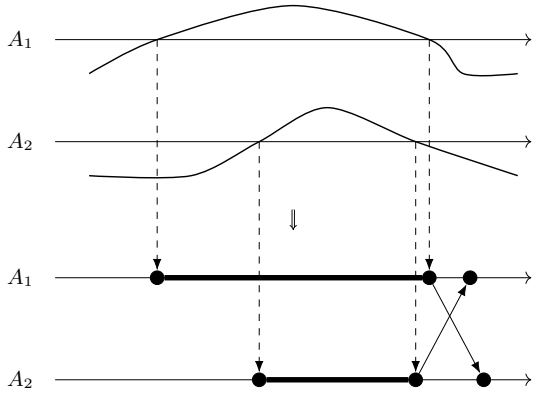


Fig. 3: A distributed signal of two agents (top) and the output of the abstractor (bottom). The abstractor marks zero-crossings as discrete root events and creates new events (dark circles) to maintain consistency.

4.2 Abstractor Description

The abstractor is described in Algorithm 1 on page 12. Its output is a stream of discrete-time events, their correct PVC values, and the relation $\rightarrow$ between them - i.e., a discrete-time distributed signal. This signal is processed by the local slicer processes as it is being produced by the abstractor.

The abstractor runs as follows. It is decentralized, meaning that there is a *local* abstractor running on each agent. Agent A_n's local abstractor maintains a buffer of discrete events, and consists of two trigger processes. The first is triggered when a root is detected (by a local zero-finding algorithm; line 1). It stores the root's information in a local buffer (for future processing). *If it is a right root*, it also sends it to the other agents. The second trigger process (line 6) is triggered when the agent *receives* a right root information from some other process, at which point it does three things: it creates a local discrete event and a corresponding relation $\rightsquigarrow$ between events (Lines 8-9), it updates events in its local buffer to see which ones can be sent to the local slicer process (described in Section 5), and then it sends them. It is clear, by construction, that $\rightsquigarrow$ is a happened-before relation: it is the subset of $\rightarrow$ needed for detection purposes.

Before an event e_n^t is sent to the slicer, it must have a PVC that correctly reflects the happened-before relation. This means that all events that happened-before e_n^t must be known to agent n, which uses them to update the PVC timestamps. This happens when events have reached agent A_n from every other agent, with timestamps that place them after e_n^t (line 11). This is guaranteed to happen by the starvation-free assumption 1.(b).

The output of a local abstractor is a stream of discrete events, so that *the output of the decentralized abstractor as a whole is a distributed discrete-time signal.* See Figure 3.

Algorithm 1: Local abstractor for agent A_n

Data: Signal of agent A_n
Result: A stream of discrete events which are roots or ϵ-offset from roots

1 **trigger** *found a* root e_n^t *at local time* t**:**
2 **add** e_n^t *info* (n, t, PVC, left or right root) to local buffer
3 **if** e_n^t *is right root*:
4 **for each** *agent* $m \neq n$:
5 **send** e_n^t *info* to agent m

6 **trigger** *received message about* right root e_m^t *from agent* A_m**:**
7 Set $t' := t + \epsilon$, where ϵ is the maximum clock skew
8 **create** *local event* $e_n^{t'}$
9 **create** *relation* $e_m^t \rightsquigarrow e_n^{t'}$ (setting the PVC for $e_n^{t'}$ appropriately)
 `/* Info for created event includes that it came from a right root` e_m^t`, not necessarily that it is a root */`
10 **add** $e_n^{t'}$ *info* (n, t', PVC, from right root) to local buffer
 `/* Ready events are those whose PVCs will not be updated anymore. See text for details. */`
11 **if** A_n *received at least one message about a* right root $e_k^{t_k}$ *from every other agent* A_k *such that* $t_k \geq t$**:**
 `/* Visit events in the buffer, forwarding ones that are ready to the slicer. */`
12 **for each** *event* e_n^s *in the local buffer*:
13 Set $\mathbf{v}_n^s[n] = s$ and $\mathbf{v}_n^s[k] = s - \epsilon$ for all $k \neq n$
14 Remove e_n^s from buffer and **send** it to local slicer

Given that all right roots are assigned discrete events by the first trigger, and given that ϵ-offset events are also created from them by the second trigger (line 8), we have the following.

Theorem 4. *All events in rightmost cuts are generated by the abstractor. Moreover, a rightmost cut of E is also a cut of the discrete signal returned by the abstractor.*

Thus the slicer process, described in the following section, can find the rightmost cuts when it processes the discrete signal. What about the leftmost satcuts? These will be handled by the slicer using the PVCs, as will be shown in the next section. Doing it this way relieves the abstractor from having to communicate the left roots between processes, thus saving on messages and their wait times.

5 The Slicer Process for Detecting Predicates

The second process in our detector is a decentralized *slicer process*, so-called to keep with the common terminology in discrete distributed systems [6]. The

slicer is decentralized: it consists of N local slicers $\mathcal{S}_n$, one per agent. The slicer runs in parallel with the abstractor and processes the abstractor's output as it is produced. Recall that the abstractor's output consists of a stream of discrete events, coming from the N agents. These events are either roots or ϵ-offset from roots. If an event is a left root or ϵ-offset from a left root, we will call it a left event. We define right events similarly. We will write F_n for those events, output by the abstractor, that occurred on A_n.

Every slicer $\mathcal{S}_n$ maintains a *token* T_n, which is a constant-size data structure to keep track of satcuts that contain A_n events. Specifically, for every event e_n^t in F_n, the token T_n is forwarded between the agents, collecting information to determine whether there exists a satcut that contains e_n^t. We say the slicer is trying to *complete* e_n^t. The token's updates are such that it will find that satcut if it exists, or determines that none exists; either way, it is then reset and sent back to its parent process A_n to handle the next event in F_n.

Let e_n^t be an event that the slicer is currently trying to complete. The token's updates vary, depending on whether it is currently completing a left event, or a right event. *If T_n is completing a right event*, the token is updated as follows. The token currently has a cut whose frontier contains e_n^t, which is either a satcut or not. If it is, the token has successfully completed the event and is returned to A_n to handle the next event in F_n. If not, then by the property of *regular* predicates [4], there exists a *forbidden event* e_m^s on the frontier of the cut which either prevents the cut from being consistent or from satisfying the predicate. T_n is sent to the process A_m containing this forbidden event. T_n's so-called target event, whose inclusion may give T_n a satcut, is the event on A_m following the forbidden e_m^s. If the token does not find a next event following e_m^s, then the token is kept by $\mathcal{S}_m$ until it receives the next event from the abstractor (which is guaranteed to happen under the starvation-free assumption). After the token retrieves the next event, the updates to the token and progression of $\mathcal{S}_n$ then follow the CGNM slicer [4]. Space limitations make it impossible to describe the CGNM slicer here, and we refer the reader to the detailed description in [4].

If handling a left event, the token is updated as follows. First, as before, T_n is sent to the process A_m which generates the forbidden e_m^s – i.e., which prevents T_n from completing e_n^t. T_n's target event may not be the next event on that process following e_m^s: that's because if e_n^t is a left root, there may exist a left event $e_m^{t-\epsilon}$ on A_m which is part of a continuous-time leftmost satcut (by Definition 3), but which was not created by the abstractor. In this case, if the token were to follow the updates for a right event, it would skip a potential satcut. Instead, the slicer $\mathcal{S}_m$ will create this event: namely, if $\mathcal{S}_m$ sees a new event $e_m^{s'}$ where $s' > t - \epsilon$, it knows that $e_m^{t-\epsilon}$ has not and will not show up (will not be produced by the abstractor) because messages are FIFO. The slicer at this point creates the new event $e_m^{t-\epsilon}$. This is valid since in continuous-time, by definition, every moment has a corresponding event on every agent. Once the token retrieves this created $e_m^{t-\epsilon}$ as its new target, the updates to the token and progression of $\mathcal{S}_n$ follow the CGNM slicer [4], similarly to the right event scenario.

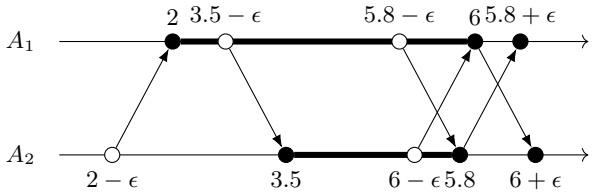


Fig. 4: Example of subsection 5.1. Bold intervals are where the local signals are non-negative. The happened-before relation is illustrated with solid arrows. The predicate is $\phi = (x_1 \geq 0) \wedge (x_2 \geq 0)$. Solid circles represent discrete events returned by the abstractor; hollow circles are those created by the slicers. The leftmost satcut of this example is $[3.5-\epsilon, 3.5]$ and the rightmost is $[6, 5.8]$.

Correctness of $\mathcal{S}$. We will show that all extremal cuts of the continuous-time signal are included in the discrete lattice of satcuts of the discrete signal. Since the CGNM slicer computes the discrete lattice, this means in particular that it computes the extremal cuts that are in it. From these extremal cuts, we can then recover the continuous-time satcuts by Lemma 1.

Theorem 5. *Our slicer returns all extremal cuts.*

We give the space and time complexity of the overall detector. Since this is an online detector which runs forever (as long as the system is alive), we must fix a time interval for the analysis.

Theorem 6. *The time complexity for each agent is $O(2RN)$, where R is the number of right roots in the given analysis interval. The detector consumes $O(N^3)$ memory to store the tokens. If roots are uniformly distributed, then the local buffers of the abstractor and slicer grow at the most to size $O(N^2)$.*

Finally, there is no bound on detection delay, since we don't assume any bounds on message transmission time. Assuming some bound on transmission delay yields a corresponding bound on detection delay.

5.1 Worked-out example

We now work through an example execution of the detector on Figure 4. We focus on agent A_2, its abstractor $\mathcal{A}_2$, slicer $\mathcal{S}_2$ and its token T_2.

1. Agent A_2 encounters a left root in the signal at local time 3.5. This information is forwarded to the abstractor.
2. The abstractor $\mathcal{A}_2$ adds the new root to its buffer with a PVC $=[3.5-\epsilon, 3.5]$.
3. A_2 finds a right root in the signal at local time 5.8 and forwards it to $\mathcal{A}_2$.
4. The abstractor sends the root information to agent A_1. It then adds this root to its buffer with a PVC timestamp of $[5.8-\epsilon, 5.8]$.
5. Abstractor $\mathcal{A}_2$ receives a message from A_1 about a right root at A_1's local time 6. Note that this is the first knowledge A_2 has about anything that is occurring on A_1, even though A_1 has already found a left root.

6. $\mathcal{A}_2$ uses A_1's message to create a new local event at $6+\epsilon$ with PVC $[6, 6+\epsilon]$.
7. $\mathcal{A}_2$ also adds this new local event to its buffer. Since all messages are FIFO, A_2 knows that there will be no new messages which will create events before $6+\epsilon$. Thus, it can remove both of the events 3.5 and 5.8 from the buffer and forward them to its local slicer $\mathcal{S}_2$. At this point both of A_1's events have been forwarded to *its* slicer, although A_2 has no knowledge of this.
8. The slicer $\mathcal{S}_2$ receives an event with a PVC $[3.5-\epsilon, 3.5]$. Token T_2 is waiting for the next event, so it adds this event to its potential cut.
9. The token is processed with its new potential cut. The cut is found to be inconsistent since T_2 has no information about any A_1 events.
10. The token's target is set to be $3.5-\epsilon$ on A_1 and the token is sent to A_1.
11. A_1 receives T_2. It walks through its local events 2 and 6 and determines that T_2's target event is between the two.
12. $\mathcal{S}_1$ creates a new event $e_1^{3.5-\epsilon}$ and notes that $x_1(3.5-\epsilon) \geq 0$.
13. Token T_2 incorporates the new event to its potential cut. The new potential cut is consistent and satisfies the predicate. It is then sent back to A_2.
14. A_2 receives T_2. T_2 indicates a satisfying cut, which the agent outputs as a result. It then advances T_2 to its next event at time 5.8.
15. T_2 has the current cut of $[3.5-\epsilon, 5.8]$. This is not consistent, so it is given the target $5.8-\epsilon$ on A_1. It is then sent to A_1.
16. A_1 receives the token. $\mathcal{S}_1$ walks through its local events and finds that the token's target is between the left root and the right root.
17. $\mathcal{S}_1$ creates a new event at $5.8-\epsilon$ and notes that $x_1(5.8-\epsilon) \geq 0$.
18. The token adds the event to its potential cut. It finds that its new potential cut is consistent and satisfies the predicate. It is then sent back to A_2.
19. A_2 receives T_2 and outputs the satcut. The algorithm then continues with new events as they occur.

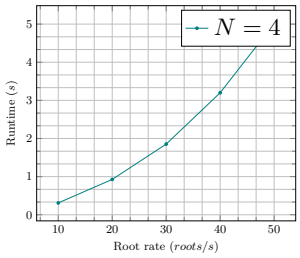
Through this example, agent A_2 discovered the satcuts $[3.5-\epsilon, 3.5]$ and $[5.8-\epsilon, 5.8]$. The first is the leftmost satcut of the interval of satcuts. A_1 discovered an additional satcut $[6, 6-\epsilon]$. Joining this satcut with A_2's second satcut returns a result of $[6, 5.8]$, which is the rightmost satcut of the interval of satcuts.

6 Case Studies and Evaluation

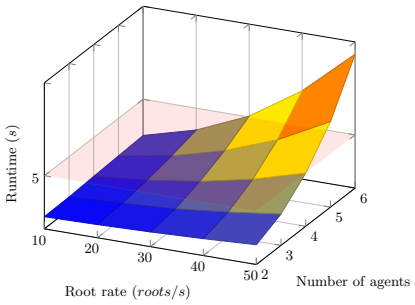
We implemented our detection algorithm and ran experiments to 1) illustrate its operation, and 2) observe runtime scaling with number of agents and with average rate of events. The detector was implemented in Julia for ease of prototyping, but future versions will be in C for speed. All experiments are replicated to exhibit 95% confidence interval. Experiments were run on a single thread of an Ubuntu machine powered by an AMD Ryzen 7 5800X CPU @ 3.80GHz. Code can be found at https://github.com/sabotagelab/phryctoria.

We consider two sources of data: the first is a set of N synthetically generated signals, $N = 1...6$. Each signal has a $5s$ duration, and is generated randomly while ensuring an average root rate of μ_n. That is, on average, μ_n roots exist

in every second of signal x_n. For the second source of data, we use the Fly-by-Logic toolbox [17] to control up to 6 simulated UAVs (i.e., drones) performing various reach-avoid missions. Their 3-dimensional trajectories are recorded over 6 seconds. We monitor the predicate "All UAVs are at a height of at least $10m$ simultaneously". Maximum clock skew ϵ is set to 0.05s.



(a) Runtime vs root rate on 4 synthetic signals.



(b) Online monitoring. The red horizontal plane indicates the runtime threshold (namely, 5s) below which it is possible to do online detection.

Fig. 5: Runtime vs root rate and N on synthetic data.

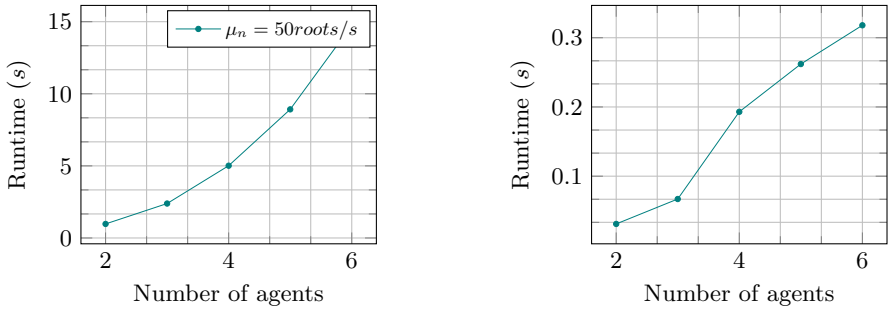
Effect of root rate (μ_n) on runtime. We use 4 synthetic signals of $5s$ duration, and measure the detection runtime as the root rate for all signals is varied between $10 roots/s$ and $50 roots/s$. Figure 5a shows the results. Naturally, as μ_n increases, so does the runtime due to having to process more tokens.

Online detection. We want to identify when it is possible for us to perform online detection with the Julia implementation, i.e. such that the detector finishes before the end of the signal being processed. To this end, we use the synthetic signals of duration 5s and vary both root rate and number of agents. Figure 5b shows the results: all combinations of root rates and number of agents with runtimes under the threshold of 5s can be performed online with the hardware setup used for these experiments.

Effect of number of agents on runtime. Figure 6 shows the effect of number of agents N on runtime. As expected, the runtime increases with N.

7 Conclusion

We have defined the first decentralized algorithm for continuous-time predicate detection in partially synchronous distributed CPS. To do so we analyzed the structure of satisfying consistent cuts for conjunctive predicates, introduced a new notion of clock, and modified a classical discrete-time predicate detector.



(a) Detection of synthetic signals at 50 roots/s

(b) Detection of UAV signals

Fig. 6: Runtime vs number of agents.

References

1. Basin, D., Klaedtke, F., Zălinescu, E.: Failure-aware runtime verification of distributed systems. In: 35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015). vol. 45, pp. 590–603. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2015)
2. Bataineh, O., Rosenblum, D.S., Reynolds, M.: Efficient decentralized ltl monitoring framework using tableau technique. ACM Transactions on Embedded Computing Systems (TECS) **18**(5s), 1–21 (2019)
3. Bauer, A., Falcone, Y.: Decentralised ltl monitoring. Formal Methods in System Design **48**, 46–93 (2016)
4. Chauhan, H., Garg, V.K., Natarajan, A., Mittal, N.: A distributed abstraction algorithm for online predicate detection. In: 2013 IEEE 32nd International Symposium on Reliable Distributed Systems. pp. 101–110. IEEE, Braga, Portugal (2013)
5. Cooper, R., Marzullo, K.: Consistent detection of global predicates. ACM SIGPLAN Notices **26**(12), 167–174 (1991)
6. Garg, V.: Elements of Distributed Computing. John Wiley & Sons (2002)
7. Garg, V.K.: Predicate detection to solve combinatorial optimization problems. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. pp. 235–245 (2020)
8. Garg, V.K., Mittal, N.: On slicing a distributed computation. In: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001. pp. 322–329. IEEE Computer Society (2001). https://doi.org/10.1109/ICDSC.2001.918962, https://doi.org/10.1109/ICDSC.2001.918962
9. Koll, C., Momtaz, A., Bonakdarpour, B., Abbas, H.: Decentralized predicate detection over partially synchronous continuous-time signals. Tech. rep., Oregon State University (2023), http://www.houssamabbas.com/wp-content/uploads/2023/08/RV_23_DMon-1.pdf
10. Kshemkalyani, A., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press (2011)
11. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (7 1978). https://doi.org/10.1145/359545.359563, https://doi.org/10.1145/359545.359563

12. Mattern, F., et al.: Virtual time and global states of distributed systems. Univ., Department of Computer Science, D 6750 Kaiserslautern, Germany (1989)
13. Mills, D., Martin, J., Burbank, J., Kasch, W.: Network time protocol version 4: Protocol and algorithms specification. Tech. rep., Internet Engineering Task Force (2010)
14. Momtaz, A., Abbas, H., Bonakdarpour, B.: Monitoring signal temporal logic in distributed cyber-physical systems. In: Proceedings of the ACM/IEEE 14th International Conference on Cyber-Physical Systems (with CPS-IoT Week 2023). p. 154–165. ICCPS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576841.3585937, https://doi.org/10.1145/3576841.3585937
15. Momtaz, A., Basnet, N., Abbas, H., Bonakdarpour, B.: Predicate monitoring in distributed cyber-physical systems. In: International Conference on Runtime Verification. pp. 3–22. Springer, Online (2021)
16. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of ltl specifications in distributed systems. In: 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 494–503. IEEE (2015)
17. Pant, Y.V., Abbas, H., Mangharam, R.: Smooth operator: Control using the smooth robustness of temporal logic. In: 2017 IEEE Conference on Control Technology and Applications (CCTA). pp. 1235–1240. IEEE (2017)
18. Sen, A., Garg, V.K.: Detecting temporal logic predicates in distributed programs using computation slicing. In: Principles of Distributed Systems: 7th International Conference, OPODIS 2003, La Martinique, French West Indies, December 10-13, 2003, Revised Selected Papers 7. pp. 171–183. Springer (2004)
19. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings. 26th International Conference on Software Engineering. pp. 418–427. IEEE (2004)