Exploiting Probabilistic Trace Expressions for Decentralized Runtime Verification with Gaps

Davide Ancona[1,†], Angelo Ferrando[1,†] and Viviana Mascardi[1,*,†]

[1] *DIBRIS, University of Genova, Via Dodecaneso 35, Genova 16146, Italy*

Abstract

Multiagent Systems (MASs) are distributed systems composed by autonomous, reactive, proactive, heterogeneous communicating entities. In order to dynamically verify the behavior of such complex systems, a decentralized solution able to scale with the number of agents is necessary. When, for physical, infrastructural, or legal reasons, the monitor is not able to observe all the events emitted by the MAS, gaps are generated. In this paper we present a runtime verification decentralized approach to handle observation gaps in a MAS.

Keywords

Decentralized Runtime Verification, Probabilistic Trace Expressions, Observation Gaps, Multiagent Systems, Agent Interaction Protocols

1. Introduction and Motivations

Distributed Runtime Verification (DRV) is a relatively new research sub-field aimed at designing fault-tolerant distributed algorithms that monitor other distributed algorithms, with the end goal of developing lightweight software systems that are more efficient that traditional verification techniques [1, 2]. The literature on DRV is almost limited [3, 4, 5, 6, 7, 8] and becomes even more limited when we consider DRV of a special kind of systems: multiagent systems (MASs [9]). In the MAS area, in fact, we are only aware of our own previous works [10, 11, 12].

Another sub-field which is raising more and more attention in the RV area concerns partial observability of the monitored events which can cause gaps in the event traces [13, 14, 15, 16]. Also in this case, when we consider MASs as the target system of the verification activity, we find very few works, all related with norm monitoring [17, 18].

This paper addresses the two issues above, decentralized runtime verification of partially observable systems, in a MAS context. The findings presented in this work can be

CILC 2022: 37th Italian Conference on Computational Logic, June 29 – July 1, 2022, Bologna, Italy

[*] Corresponding author.

[†] These authors contributed equally.

davide.ancona@unige.it (D. Ancona); angelo.ferrando@unige.it (A. Ferrando); viviana.mascardi@unige.it (V. Mascardi)

https://person.dibris.unige.it/ancona-davide/ (D. Ancona); https://angeloferrando.github.io/website/ (A. Ferrando); https://person.dibris.unige.it/mascardi-viviana/ (V. Mascardi)

0000-0002-6297-2011 (D. Ancona); 0000-0002-8711-4670 (A. Ferrando); 0000-0002-2261-9926 (V. Mascardi)

generalized and applied to other kinds of systems, but – for presentation purposes – we concentrate our investigation on MASs.

The main source of inspiration for our work is the paper by Stoller et al. [14], where the authors introduce *runtime verification with state estimation.* With respect to a more standard RV approach, they are interested in checking system executions (traces) containing *gaps.* A gap represents the absence of information in the trace of observed events and corresponds to an execution point where the monitor knows that the system emitted some event, but does not know which one. In offline RV gaps in the trace logs are due to the process of sampling observed events in order to reduce the monitoring overhead. Gaps can also be met in online RV, where the system behavior is analyzed while the system is running and problems with the infrastructure, privacy and legal issues that prevent the monitor to observe some kind of events, faults in the monitor observation capabilities, may generate gaps. Although the problems raised by online and offline RV with gaps share many similarities, the online setting is much more challenging. Each time a gap is perceived, the monitor must make guesses on the possible actual events that the gap represents and save all the states generated by these guesses. A possibly huge logical tree-like structure with states as nodes, and moves from states to states as edges, represents the open possibilities[1]. In offline RV, this logical tree-like structure can be explored following a depth-first search, requiring a limited amount of memory. If the RV takes place online, its exploration must follow a breadth-first strategy, with much more space needed to save the states, as the final trace of events is unknown and the levels of the structure are generated and explored at the same time. In order to cope with the state space explosion due to guesses in the online RV scenario, we propose to *decentralize* the monitoring activity.

RV decentralization is a very natural choice when the system under monitoring is a MAS, which is *distributed by definition*, and may improve *efficiency*, as the verification process can be spread on different machines improving performance; *scalability*, as under some conditions depending on the protocol [12, 10] it is possible to associate one monitor with each agent in the MAS, keeping under control the RV complexity even when the number of agents grows; *feasibility*, as for physical/logical/legal reasons one single monitor might not be able to observe all the events generated by the MAS.

The feature that is usually subject to verification (both static and dynamic) in a MAS is its *communicative behavior* [19, 20, 21, 22, 23, 24, 25, 26, 27]. With respect to [14], in this work we do not aim at verifying temporal properties. Rather, we want to check the conformance of the MAS actual communicative behavior to an Agent Interaction Protocol (AIP) that models the allowed interactions among agents, under the hypotesis that some interactions could not be observed. The research question we address is thus *how to evaluate the probability that a MAS satisfies an AIP, in the presence of gaps.*

In a recent paper [28] we introduced Probabilistic Trace Expressions (PTEs) and the theory behind them. In this work we take a more pragmatical perspective and we show

[1] In the remainder we will use the term “branch” to denote paths in this logical structure, and we will sometime use “states” meaning “the final states of all the possible branches”, when this does not generate confusion.

how to use PTEs for decentralized RV of AIPs within MASs with gaps.

2. Background

Probabilistic Trace Expressions. Trace expressions [29, 30, 31, 32, 33, 34, 35, 36, 37] are based on the notions of *event* and *event type*. We denote by $\mathcal{E}$ the fixed universe of events subject to monitoring. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$, and a trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$. Trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), each specifying a subset of events in $\mathcal{E}$. A trace expression $\tau \in \mathcal{T}$ represents a set of possibly infinite event traces, and is defined on top of the following operators:

• ϵ (empty trace), denoting the singleton set $\{\epsilon\}$ containing the empty event trace ϵ.

• $\vartheta{:}\tau$ (*prefix*), denoting the set of all traces whose first event e matches the event type ϑ, and the remaining part is a trace of τ.

• $\tau_1{\cdot}\tau_2$ (*concatenation*), denoting the set of all traces obtained by concatenating the traces of τ_1 with those of τ_2.

• $\tau_1{\wedge}\tau_2$ (*intersection*), denoting the intersection of the traces of τ_1 and τ_2.

• $\tau_1{\vee}\tau_2$ (*union*), denoting the union of the traces of τ_1 and τ_2.

• $\tau_1|\tau_2$ (*shuffle*), denoting the set obtained by shuffling the traces of τ_1 with the traces of τ_2.

Trace expressions support recursion through cyclic terms expressed by finite sets of recursive syntactic equations, as supported by modern Prolog systems.

A probabilistic trace expression is a trace expression where event types have a probability associated with them [28], and its modeling and semantics are also implemented in Prolog (SWI-Prolog, see the code available here, https://vivianamascardi.github.io/Software/PTE.pl). PTEs are suitable to manage guesses in the presence of observation gaps; in order for this management to work, we assume that *each gap represents one single unobserved event*.

As an example, the probabilistic trace expression

$$\tau = e_1[0.2]{:}\tau_1{\vee}e_2[0.8]{:}(\tau_2|\tau_3)$$

represents the protocol where we can accept the event e_1 with probability 0.2, or, the event e_2 with probability 0.8. If we consume the event e_1, we go to the new state τ_1, while, if we consume e_2, we go to a state where we can have all possible interleaving of τ_2 and τ_3. If there is a gap in the monitoring activity and the monitor is not able to observe which event took place, it can nevertheless make its guesses which involve e_1 and e_2, associate a probability with each of them, and keep both possibilities.

Like a "normal" trace expression, a probabilistic trace expression τ can be seen as the current state of a protocol that started in some initial state τ_{init} and reached τ after n events $O_1 ... O_n$ took place. These events moved τ_{init} to τ through intermediate states τ_{q1}, τ_{q2}, ... , $\tau_{qn} = \tau$. If we denote with $\tau \overset{O}{\rightarrow} \tau'$ the transition from state τ to state τ' due to the event O taking place and being observed, we may write

$\tau_{init} \overset{O_1}{\rightarrow} \tau_{q1} \overset{O_2}{\rightarrow} \tau_{q2} \overset{O_3}{\rightarrow} \tau_{q3} ... \overset{O_n}{\rightarrow} \tau_{qn}$, where $\tau_{qn} = \tau$.

In order to properly manage probabilities, it is convenient to associate with τ – in an explicit and easily computable way – the probability of the protocol to have reached τ starting from τ_{init} and having observed $O_1 ... O_n$.

We define a "probabilistic trace expression state" the triple consisting of a trace expression τ, a sequence of events $O_1 ... O_n$ observed before reaching τ, and the probability π_τ that the protocol reached τ. We represent the state with the notation $\langle \tau, \pi_{tr}, O_1 ... O_n \rangle$.

Decentralized MAS Monitoring with DecAMon. In [10] we presented the DecAMon algorithm to decentralize agent interaction protocols modeled using trace expressions. There, we defined the notion of "monitoring safe" partition. A partition can be used to drive the distribution process. To decentralize the monitoring activity, we project the global AIP onto each subset of agents belonging to the partition, where by "projection" we mean that we maintain only the interactions involving agents in the chosen subset. In general, not all the partitions can be used for the RV decentralization. A partition that can be used to decentralize the RV of a protocol is called "monitoring safe" and the algorithm presented in [10] generates all the monitoring safe partitions for a given AIP.

Since under the conditions considered in this paper we may observe gaps, we could not have only one single state representing the current situation of the protocol, like it happens in our previous works; instead, we have to maintain all the states that may be possibly reached "via the gaps". As already anticipated, each state can be represented as a tuple $\langle \tau, \pi, evs \rangle$, where τ is the PTE representing the current state of the protocol and π is the joint probability that the sequence of events evs is compliant with τ [28].

Let us name M_0 the set of possible initial states of the monitor (as there may be more than one). The number 0 stands for the $0th$ iteration, since at the beginning we have not consumed any event yet. We can first run DecAMon on the global AIP to find a good set of monitoring safe partitions and, after that, we can use one of them to project the τs in M_0 onto the subsets of the agents. Once we have obtained the distributed versions of the initial τs via projection, we can generate one monitor for each partition, and decentralize the RV.

The combination of decentralization and lack of information calls for a synchronized management of gaps. Since each monitor has a different state representing its current protocol evolution, when there is an observation gap, each monitor can have different opinions about which are the correct events that might suitably "fill the gap". The local perspectives can be compared and used by the monitors to cut wrong guesses, and hence wrong states, on the basis of distributed knowledge. Despite the overhead due to synchronization, this approach may dramatically improve performance, as discussed in the next sections.

3. Handling Gaps in Decentralized RV

Gaps represent lack of information, thus a point (or points) in the event trace where the monitor does not know what event had been actually generated by the system under monitoring. In the remainder we will write that "gaps can be observed", in the sense that a monitor can realize that something went wrong and that an event was generated by the system, and not correctly observed. We also assume that, in a decentralized setting, when one monitor "observes a gap", all the monitors "observe a gap" as well. If this gap does not involve the sub-system monitored by a monitor M, the trace observed by M will contain $gap(none)$: this notation means – from M's point of view – "I am aware that some event was generated by some component of the system that I am not in charge of, and that the event was not correctly observed". From a technical viewpoint, this could be obtained by forcing one monitor to inform the others when it observes a gap. This would require some shared clock among the monitors as, in order for our algorithm to work, the gap must take place at the same time for all the monitors hence raising clock synchronization issues. Given that these issues are well known and well studied in distributed systems [38], we leave them out of our investigation. Being well studied does not mean to be easy to face. Indeed, we are aware that the need of observing all the gaps at the same time in a decentralized setting, represents a serious limitation of our framework and we are working towards alternative, and more feasible, solutions.

When a centralized monitor observes a gap, since it is the only monitor checking the event trace w.r.t. the AIP specification, it can make guesses on what the gap is and reason on its own guesses, eventually tagging some of them as wrong due to successive observations. When there are many monitors, each one monitoring a subset of the agents, and hence a sub-protocol of the global AIP, each monitor can still suppose what the observed gap is, but the reasoning on its suppositions must be shared with the others. This sharing phase among the monitors is crucial, because it allows them to cut wrong branches on the basis of what other monitors suppose, or what they are fully sure of.

Let us consider two monitors m_1 and m_2 that observe a gap. Given that the protocols driving the two monitors are different, although being derived via projection from the same global protocol, m_1 might suppose that the events admissible for filling the observed gap are e_1 and e_2, while m_2 could instead suppose that admissible events are e_2 and e_3. Both m_1 and m_2 must keep track of these possibilities in their local knowledge bases, and – so far – they do not need to share they guesses.

Let us now suppose that in the current state of m_1, in the branch where e_1 was supposed to have taken place, the only successive possible event is e_4, while in the branch for e_2 the only possible event is e_5. If, after the gap, m_1 observes e_5, it can cut the branch where the gap was associated with e_1, because e_5 would not be allowed after e_1. The gap before e_5, that could be filled in principle by e_1 and e_2, becomes bound - "without any doubt"[2] - to e_2. After having found the right value for the gap and cut one branch, m_1 informs m_2 allowing it to cut the branch where the value for that gap was guessed to be

[2] Modulo the assumption that observed events are compliant with the foreseen protocol. Gaps may inevitably generate false negatives. In this case, m_1 assumes that the gap was e_2 because this would be consistent with the successive observation of e_5 and with the protocol to be respected. If the gap were any other event, a protocol violation would have taken place and m_1 should have raised a protocol monitoring exception. Depending on the protocol, the violation could be recognized later on, or never. Suppose for example an infinite protocol where only as are allowed. A gap will be necessarily filled with a even if the actual event was b, and if the successive observed events are all as, the violation will never be discovered.

e_3. In this way, both m_1 and m_2 can continue the verification process supposing that the unobserved event represented by the gap was e_2, with some given probability due to the probability associated with e_2 in the PTE modelling the protocol.

Before presenting the decentralized monitoring algorithm, we make some considerations on the kind of gaps a monitor can observe. So far, we considered generic events. This is correct and consistent with the general approach presented in [28], but in a MAS scenario where PTEs model agent interaction protocols we can be more specific. In this scenario, in fact, the universe of events is $Msgs$, namely the universe of the possible messages among agents. Such special events can be represented as $a_1 \overset{c}{\Longrightarrow} a_2$, meaning that agent a_1 sends a message to a_2 with content c. Since messages are composed by (at least) three mandatory components, sender, receiver and content, there can be many partially instantiated gaps such as:

- $gap(a_1 \Longrightarrow a_2)$, where the content of the message is unknown;
- $gap(\_ \overset{m}{\Longrightarrow} a_2)$, where the sender is unknown;
- $gap(a_1 \overset{m}{\Longrightarrow} \_)$, where the receiver is unknown.

Although, for sake of clarity, in the sequel we consider gaps where neither the sender, nor the receiver, nor the content are known (total absence of information), all the combinations of "information holes" are possible, and partially instantiated gaps may be exploited to reduce branches due to guesses. The algorithm presented in the next section can be easily adjusted to take partially instantiated gaps into account.

Synchronizing Decentralized Gaps Management. We present the algorithm used by the decentralized monitors to synchronize the gaps management, in order to cut useless branches and check the compliance of interactions with the protocol. When an event is generated by the system, two different situations can take place.

Case 1: The event is not a gap

If the event is not a gap, each monitor that observed it can use the event for updating its local state(s). If some branches have been removed as in the previous example involving m_1 and m_2, the monitor has to inform the other monitors of the associations between gaps and events that are not admissible any longer. This phase can be reiterated until all the monitors have cut all the possible wrong brnches, and have nothing more to say. After this synchronizations stage, the monitoring process continues in the normal way.

Case 2: The event is a gap

To keep the presentation simple, we assume that gaps are observed by all the monitors at the same time. Each monitor guesses the events admissible to fill the gap, according to its local states. If the gap is partially instantiated (some of its components were correctly observed, like the sender, or the content, or both), the monitor can use this information to reduce the set of possible candidate events.

The two cases can be seen as a *reduce* and *extend* stages, respectively. When the monitor observes a fully instantiated event it can invalidate zero, one or more branches. If the invalid branches contain gaps, the monitor can also invalidate the associations between these gaps and the guessed events, and can allow the other monitors to invalidate these associations as well via communication. On the other hand, observation of gaps generates as many branches as the events that, according to the AIP, could fill the gap. We can formalize this intuition in the following way.

Given M_0 as the set of global states $\{\langle \tau_1, \pi_1, [] \rangle, ..., \langle \tau_n, \pi_n, [] \rangle\}$.

1. Distribute M_0 with respect to a given partition $P = \{\{ags_1\}, ..., \{ags_{np}\}\}$, projecting the states onto subsets of the agents involved (the function Π projects an AIP τ onto a set of agents ags removing all the events whose sender and receiver do not belong to ags), obtaining

$$M_{0,\{ags_1\}} = \{\langle \Pi(\tau_1, \{ags_1\}), \pi_1, [] \rangle, ..., \langle \Pi(\tau_n, \{ags_1\}), \pi_n, [] \rangle\}$$
$$...$$
$$M_{0,\{ags_{np}\}} = \{\langle \Pi(\tau_1, \{ags_{np}\}), \pi_1, [] \rangle, ..., \langle \Pi(\tau_n, \{ags_{np}\}), \pi_n, [] \rangle\}$$

2. Each monitor observes only the event messages involving the agents belonging to its set ags_i:
   a) if the event message is a gap, the monitor guesses what it could be and generates as many states as the possible events (*extend*);
   b) if the event message is ground, the monitor can cut branches, and in this case it communicates with other monitors the gap values that are no longer admissible (*reduce*).
3. If, after observation of an event or because of information received from other monitors, the set of possible current states for a monitor m becomes empty, m stops the monitoring process, informs all the other monitors, and they also stop monitoring. The absence of possible current states for a monitor is due to a protocol violation that took place, preventing at least one monitor to move a further step. So, the system checked does not satisfy the agent interaction protocol and the associated probability is 0.
4. Else,
   a) if there are no events left to analyze, the monitoring process ends and the resulting probability is evaluated (see after how);
   b) else, repeat from step 2.

To be clearer, in step 2, given the current event message, each monitor queries its current state following the PTE operational semantics presented in [28] in order to check if the event message is admissible or not. In the updating phase, the monitors inform the others trying to cut not admissible branches.

If the monitoring process ends without violations detected and there are no more events left to analyze, each monitor stops with at least one admissible branch. Each monitor states its own evaluation of the probability that the system's behavior satisfies the agent interaction protocol. This probability can be computed summing up all the joint probabilities contained in all the final states, corresponding to the last nodes of the admissible branches. This leads to having one estimated value for each monitor: we can adopt different strategies to summarize the final, and global, one, such as taking the smallest (largest) value among all those estimated by all the monitors, or a weighted means where weights model each monitor's trustability, or other domain-dependent strategies.

4. Example

We present a simple example helping us to show how the *extend* and *reduce* steps work. We consider a MAS involving four agents: $\{alice, bob, charlie, dave\}$. The set of events of our interest is the set of messages that these agents can use to communicate with each other.

Given the PTE

$$\tau = \tau_1 \vee \tau_2$$

$$\tau_1 = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}(bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_1 | bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon)$$

$$\tau_2 = alice \overset{msg_4}{\Longrightarrow} dave[0.3]{:}(charlie \overset{msg_5}{\Longrightarrow} dave[0.3]{:}\epsilon | bob \overset{msg_3}{\Longrightarrow} dave[0.7]{:}\tau_2)$$

We decentralize τ on each single agent, obtaining[3]:

$$M_{0,\{alice\}} = \{\langle \Pi(\tau, \{alice\}), 1, [] \rangle\} = \{\langle \tau_{alice}, 1, [] \rangle\}$$

$$M_{0,\{bob\}} = \{\langle \Pi(\tau, \{bob\}), 1, [] \rangle\} = \{\langle \tau_{bob}, 1, [] \rangle\}$$

$$M_{0,\{charlie\}} = \{\langle \Pi(\tau, \{charlie\}), 1, [] \rangle\} = \{\langle \tau_{charlie}, 1, [] \rangle\}$$

$$M_{0,\{dave\}} = \{\langle \Pi(\tau, \{dave\}), 1, [] \rangle\} = \{\langle \tau_{dave}, 1, [] \rangle\}$$

where

$$\tau_{alice} = \tau_{1_{alice}} \vee \tau_{2_{alice}}$$

$$\tau_{1_{alice}} = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}\tau_{1_{alice}}$$

$$\tau_{2_{alice}} = alice \overset{msg_4}{\Longrightarrow} dave[0.3]{:}\tau_{2_{alice}}$$

$$\tau_{bob} = \tau_{1_{bob}} \vee \tau_{2_{bob}}$$

$$\tau_{1_{bob}} = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}(bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_1 | bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon)$$

$$\tau_{2_{bob}} = bob \overset{msg_3}{\Longrightarrow} dave[0.7]{:}\tau_{2_{bob}}$$

$$\tau_{charlie} = \tau_{1_{charlie}} \vee \tau_{2_{charlie}}$$

$$\tau_{1_{charlie}} = bob \overset{msg_2}{\Longrightarrow} charlie[0.6]{:}\tau_{1_{charlie}}$$

$$\tau_{2_{charlie}} = charlie \overset{msg_5}{\Longrightarrow} dave[0.3]{:}\tau_{2_{charlie}}$$

$$\tau_{dave} = \tau_{1_{dave}} \vee \tau_{2_{dave}}$$

[3] The initial probability of each state is 1, since we do not want to influence the probability evaluation process (multiplication of probabilities).

$$\tau_{1_{dave}} = \mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave}[0.4]{:}\epsilon$$

$$\tau_{2_{dave}} = \mathit{alice} \overset{msg_4}{\Longrightarrow} \mathit{dave}[0.3]{:}(\mathit{charlie} \overset{msg_5}{\Longrightarrow} \mathit{dave}[0.3]{:}\epsilon|\mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave}[0.7]{:}\tau_{2_{dave}})$$

Let us suppose that the monitors observe a *gap* now. Each monitor moves to a new set of states corresponding to the possible values for the *gap*.

$$M_{0,\{alice\}} \overset{gap}{\to} \{\langle \tau_{1_{alice}}, 0.7, [\mathit{gap}(\mathit{alice} \overset{msg_1}{\Longrightarrow} \mathit{bob})]\rangle,$$

$$\langle \tau_{2_{alice}}, 0.3, [\mathit{gap}(\mathit{alice} \overset{msg_4}{\Longrightarrow} \mathit{dave})]\rangle,$$

$$\langle \tau_{alice}, 1, [\mathit{gap}(\mathit{none})]\rangle\} = M_{1,\{alice\}}$$

$$M_{0,\{bob\}} \overset{gap}{\to} \{$$

$$\langle(\mathit{bob} \overset{msg_2}{\Longrightarrow} \mathit{charlie}[0.6]{:}\tau_1|\mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave}[0.4]{:}\epsilon), 0.7, [\mathit{gap}(\mathit{alice} \overset{msg_1}{\Longrightarrow} \mathit{bob})]\rangle,$$

$$\langle \tau_{2_{bob}}, 0.7, [\mathit{gap}(\mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave})]\rangle,$$

$$\langle \tau_{bob}, 1, [\mathit{gap}(\mathit{none})]\rangle\} = M_{1,\{bob\}}$$

$$M_{0,\{charlie\}} \overset{gap}{\to} \{$$

$$\langle \tau_{1_{charlie}}, 0.6, [\mathit{gap}(\mathit{bob} \overset{msg_2}{\Longrightarrow} \mathit{charlie})]\rangle,$$

$$\langle \tau_{2_{charlie}}, 0.3, \mathit{gap}(\mathit{charlie} \overset{msg_5}{\Longrightarrow} \mathit{dave})\rangle,$$

$$\langle \tau_{charlie}, 1, [\mathit{gap}(\mathit{none})]\rangle\} = M_{1,\{charlie\}},$$

$$M_{0,\{dave\}} \overset{gap}{\to} \{$$

$$\langle \epsilon, 0.4, \mathit{gap}(\mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave})\rangle,$$

$$\langle(\mathit{charlie} \overset{msg_5}{\Longrightarrow} \mathit{dave}[0.3]{:}\epsilon|\mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave}[0.7]{:}\tau_{2_{dave}}), 0.3, \mathit{gap}(\mathit{alice} \overset{msg_4}{\Longrightarrow} \mathit{dave})\rangle,$$

$$\langle \tau_{dave}, 1, [\mathit{gap}(\mathit{none})]\rangle\} = M_{1,\{dave\}}$$

Since they observed a *gap*, the monitors do not know what the actual event was. Because of this, they have to generate more branches, where each branch represents a possible value for the gap. This is the *extend* step.

Let us now suppose that the monitors observe event msg_2. Since msg_2 is a ground event, everything is known about it, in particular the monitors know that its sender is *bob* and its receiver is *charlie*. Since the monitors observe only the gaps and the events that involve the agents in the partition they are in charge for, the only monitors that observe msg_2 are $M_{1,\{bob\}}$ and $M_{1,\{charlie\}}$.

By consuming msg_2, the first iteration of the algorithm leads to:

$$M_{1,\{bob\}} \overset{\mathit{bob} \overset{msg_2}{\Longrightarrow} \mathit{charlie}}{\to} \{$$

$$\langle \tau_1|\mathit{bob} \overset{msg_3}{\Longrightarrow} \mathit{dave}[0.4]{:}\epsilon, 0.42, [\mathit{gap}(\mathit{alice} \overset{msg_1}{\Longrightarrow} \mathit{bob}), \mathit{bob} \overset{msg_2}{\Longrightarrow} \mathit{charlie}]\rangle$$

$$\} = M_{2,\{bob\}}$$

$$M_{1,\{charlie\}} \overset{bob \overset{msg_2}{\Longrightarrow} charlie}{\rightarrow} \{$$

$$\langle \tau 1_{charlie}, 0.36, [gap(bob \overset{msg_2}{\Longrightarrow} charlie), bob \overset{msg_2}{\Longrightarrow} charlie] \rangle,$$

$$\langle \tau 1_{charlie}, 0.6, [gap(none), bob \overset{msg_2}{\Longrightarrow} charlie] \rangle$$

$$\} = M_{2,\{charlie\}}$$

It is interesting to analyze what happened in $M_{2,\{bob\}}$, where the *reduce* step took place. In fact, the ground event msg_2 makes the other two branches not valid anymore. More in detail, the second branch was $\langle \tau 2_{bob}, 0.7, [gap(msg_3)] \rangle$, and $\tau 2_{bob}$ does not accept the event msg_2 and cannot move to a new state. In the same way, the PTE in the third branch $\langle \tau_{bob}, 1, gap(none) \rangle$ is τ_{bob}, and τ_{bob} cannot accept the event msg_2 either. Even though this information seems important for monitor $M_{2,\{bob\}}$ only, it is actually of interest also for the other monitors. In fact, it allows all of them to know "without any doubt" that the only event that can be associated with the first gap is msg_1, since it is the gap value associated with the only possible branch of $M_{2,\{bob\}}$. The monitor $M_{2,\{bob\}}$ can inform the other monitors that the only admissible value for the gap is msg_1. The monitors' new states become:

$$M_{2,\{charlie\}} = \{\langle \tau 1_{charlie}, 0.6, [gap(none), bob \overset{msg_2}{\Longrightarrow} charlie] \rangle\}$$

$$M_{1,\{alice\}} = \{\langle \tau 1_{alice}, 0.7, [gap(alice \overset{msg_1}{\Longrightarrow} bob)] \rangle\}$$

$$M_{1,\{dave\}} = \{\langle \tau_{dave}, 1, [gap(none)] \rangle\}$$

This example shows how the knowledge of a monitor can have a positive impact on the knowledge of the other monitors. In general, this positive impact can be obtained any time one monitor discovers that one branch is no longer valid and can hence invalidate the associations of events with gaps therein. This information may trigger many communication iterations among the monitors, because, when one monitor is updated it can also "invalidate one branch" and the related gap-events associations, and may need to inform the others of some association which is no longer possible. In the previous example, one single iteration was enough.

As we already anticipated, the proposed approach may lead to false negatives, due to an optimistic approach of the monitors that stubbornly assume that observed events are compliant with the protocol, if there is just one possibility left to make such an assumption. Also in this example, the monitors gave the correctness of the ground event msg_2 (the second event observed) for granted. But let us suppose that the actual event masked by the *gap* was not msg_1, but msg_4, and that the successive message msg_2 was sent from *bob* to *charlie* by mistake and did not comply with the protocol. In this scenario, since the monitors do not know for sure what the first *gap* was, it is reasonable to consider msg_2 a valid message and hence cut the branch where the gap has been supposed to be msg_4. This is a problem intrinsically related to the state estimation approach, since until it is acceptable to observe an event in a state, the monitors keep track of the related branch. Only when a monitor, observing an event, loses all its branches it can conclude

Table 1
Average time of the centralized and decentralized algorithms; “sh. PTE” stands for “shuffled sub-PTE”.

# sh. PTEs	# agents for sh. PTE	# operations for sh. PTE	Centralized [sec]	Decentralized [sec]
10	10	20	6.64	1.26
10	10	15	8.26	1.04
10	5	20	9.85	1.49
10	5	15	9.92	1.28
10	15	15	14.86	1.23
10	5	10	18.35	1.08
10	15	10	20.25	1.61
10	10	10	29.59	1.98
15	5	15	93.34	2.73
15	15	10	116.61	3.56
10	15	20	126.31	25.32
15	10	10	283.70	4.14
15	5	10	349.30	2.23
20	10	10	355.90	3.99
15	5	20	363.67	5.83
20	5	15	558.59	9.28
20	5	20	801.37	7.82
15	20	10	952.43	12.36
20	5	10	1223.85	10.64
20	15	10	1340.29	9.57
20	20	10	1727.26	2.89

that a protocol violation took place because some wrong assumption on gaps – confirmed by successive observations – had been made in the past. This delay in the error detection, which could also be infinite, can be reduced introducing a threshold on the probability that a branch must have to be considered valid. In this way, if after observing an event the probability associated with a branch becomes lower than a chosen threshold, the monitor can cut that branch and make error detection possibly quicker.

5. Experimental results

In our experiments we have considered the four following features:

1. the number of agents involved in the MAS we want to verify at runtime;
2. the number of *shuffled sub-PTEs* due to shuffle operators | in the AIP: we name shuffled sub-PTE each portion of the PTE composed via a |, so for example $\tau_3 = alice \overset{msg_1}{\Longrightarrow} bob[0.7]{:}\epsilon \mid bob \overset{msg_3}{\Longrightarrow} dave[0.4]{:}\epsilon$ consists of 2 *shuffled sub-PTEs*; we point out that when decentralizing the monitoring, we can associate one different monitor with each shuffled sub-PTE, as shuffled sub-PTE are independent one from the other and can be monitored in a fully decentralized way;
3. the number of operators for each shuffled sub-PTE in the AIP;
4. the number of gaps contained in the analyzed traces.

In Table 1, we report the results of our experiments. For each row, we keep the number of shuffled sub-PTE, agents and operators fixed, while we change the length of the traces and the percentage of gaps inside each trace. For each row we executed many different runs and we have measured the total time required for recognizing the set of 300 randomly generated traces. We changed the number of gaps contained inside the traces and we tested both the centralized [28] and the decentralized algorithms. In the following, we reported the graphics obtained from such executions.

Concerning the figures, *the traces used in our experiments contain only gaps* (namely, we run experiments in the worst possible scenario), so the algorithm makes only expansions and never reductions. We chose traces with only gaps to stress the algorithms as much as possible. In real scenarios gaps should be the exceptions, and perfectly observable events the norm.

In Figures 1 and 2, both the centralized and the decentralized algorithms seem to show linear complexity with respect the number of the agents involved, even if the decentralized algorithm has better performances.

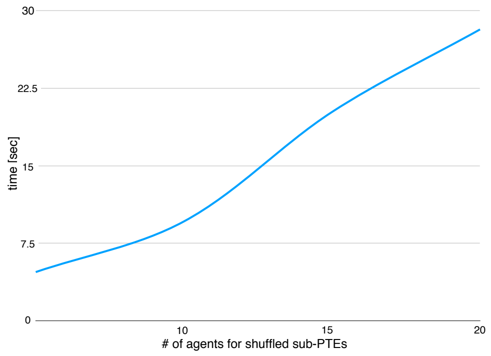

Figure 1: Centralized algorithm: changing number of agents.

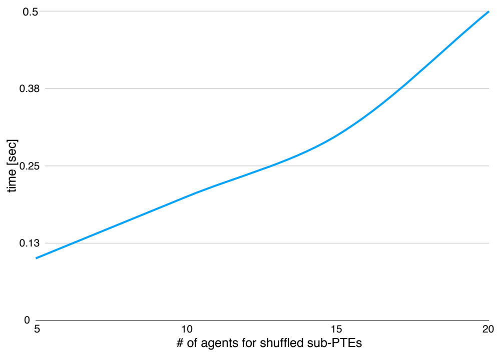

Figure 2: Decentralized algorithm: changing number of agents.

In Figures 3 and 4, we can observe that the complexity of the centralized algorithm seems to grow in a quadratic way, while the decentralized one seems to grows linearly. This can be explained by the decentralization of the monitoring of shuffled sub-PTEs, as if we add one operator to each shuffled sub-PTE, the monitor in charge for that shuffled sub-PTE will need to manage one more operator only, whereas the centralized monitor will cope with as many new operators as the shuffled sub-PTEs in the trace expression. We point out that we use “seems to” to reflect that the complexities emerging from the figures have not been computed on the basis of the algorithm, but have been estimated on the basis of the experiments, and the behaviour in situations involving a limited number of agents, operators, shuffled sub-PTEs, might not be the actual asymptotic behaviour of the algorithm.

In Figures 5 and 6, we can appreciate the real advantages of decentralization, as – from the figures – it seems that we have an exponential complexity for the centralized algorithm and a pseudo-quadratic complexity for the decentralized one. We emphasise that in the decentralized case (Figure 6) we were able to run experiments with 40 shuffled sub-PTEs, while in the centralized case we had to stop with half shuffled sub-PTEs, and with an execution time hundred times higher. The number of shuffled sub-PTEs is indeed the feature which most impacts the algorithms performance, and this in not a surprise; intuitively, when we add a new shuffled sub-PTE we have to interleave it with all the already existent shuffled sub-PTEs. In the centralized case, this brings to a

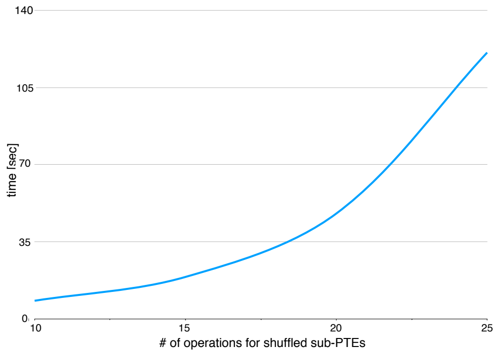

Figure 3: Centralized algorithm: changing number of operators.

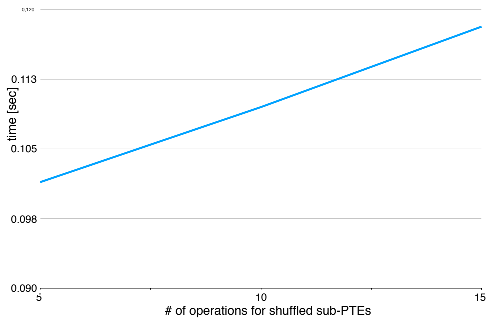

Figure 4: Decentralized algorithm: changing number of operators.

state explosion, while in the decentralized one, since we can decentralize the monitoring of each shuffled sub-PTEs, we simply have to add a new monitor. In this way, we can avoid the state explosion, even if the presence of a new monitor increases the exchange of messages among the monitors needed to synchronize information about gaps.

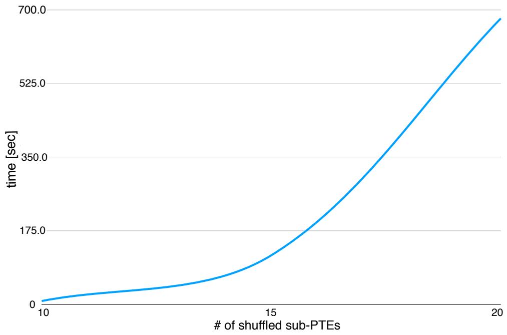

Figure 5: Centralized algorithm: changing number of shuffled sub-PTEs.

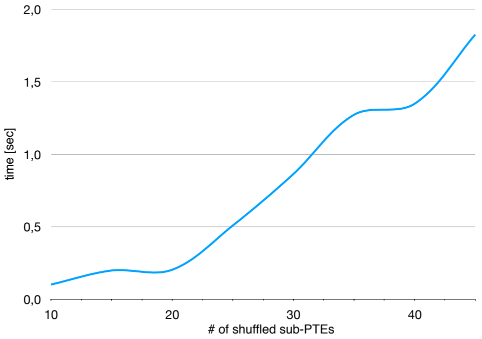

Figure 6: Decentralized algorithm: changing number of shuffled sub-PTEs.

So far, our experiments are simulated: although trace expressions and the RML language[4] that spun from them [37] have been adopted to model and dynamically verify behavioural patterns involving interaction among Jason and JADE agents [39], robotic systems [40], IoT systems [41] and programs developed in Node.js and Node-RED [37], their probabilistic extension, PTEs, was not tested on real setting yet.

The code for running our experiments can be found in the PTE repository on GitHub, https://vivianamascardi.github.io/Software/PTE.pl.

[4] https://rmlatdibris.github.io/, accessed on June 2022.

By calling the `generate_mas` goal, we can generate one simulated PTE with some features, for example `generate_mas(5, 3, 4, Partition, T)` unifies `T` with a PTE with 5 branches, 3 randomly generated agents involved in each branch, and 4 randomly selected operators (such as shuffle, union, etc) for each branch, and `Partition` with a partition of `T` into sub-PTEs to be monitored in a decentralized way. Then, the `T` and `Partition` variables unified with ground temrs can be used as arguments of the `create_output_file` goal that generates a csv file containing all the data needed for making the experiments presented in this section.

In `create_output_file(ID, T`, `MaxLength`, `NTests`, `MinProbNoise`, `MaxProbNoise`, `MinProbMsgNoise`, `MaxProbMsgNoise`, `Partition)`, `ID` is the csv file name to be generated, `T` and `Partition` must be unified with a ground PTE and with a partition into sub-PTEs, respectively, by calling `generate_mas`, `MaxLength` is the maximum length of the trace to analyze, `NTests` is the number of tests to be repeated, to have more robust and reliable results, `MinProbNoise` and `MaxProbNoise` define the probability range to have gaps in the generated and analyzed trace, `MinProbMsgNoise` and `MaxProbMsgNoise` define the probability range to have gaps in the observed message.

6. Conclusions and Future Work

In this paper we presented a distributed approach to runtime verification where we may lack some pieces of information about observed events. With respect to standard runtime verification, the state estimation approach allows us to be more reliable, especially in scenarios where partial or total absence of information is frequent.

For the sake of clarity, we considered only totally uninstantiated gaps. This choice has been made to make the development of monitors easier. Naturally, the presence of part of information about the event could be used by the monitors in order to cut useless branches. We will extend our implementation to cope with partially instantiated gaps.

Another future work will be to consider a threshold in order to cut branches that are unreasonable to maintain, as the probability to be correct is too low. Fixed a threshold, a monitor will be able to remove all the branches with a joint probability associated with them lower than the chosen threshold. This will bring the advantage of anticipating the error detection and to prune useless branches related to unreasonable possibilities.

References

[1] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, C. Travers (Eds.), Bertinoro Seminar on Distributed Runtime Verification, May 2016, Available from `http://www.labri.fr/perso/travers/DRV2016/`, 2016.

[2] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, C. Travers, Challenges in fault-tolerant distributed runtime verification, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016. Proceedings, Part II, Springer, 2016, pp. 363–370.

[3] P. Fraigniaud, S. Rajsbaum, M. Roy, C. Travers, The opinion number of set-agreement, in: M. K. Aguilera, L. Querzoni, M. Shapiro (Eds.), Principles of Distributed Systems: 18th International Conference, OPODIS 2014. Proceedings, Springer, 2014, pp. 155–170.

[4] P. Fraigniaud, S. Rajsbaum, C. Travers, On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems, in: B. Bonakdarpour, S. A. Smolka (Eds.), Runtime Verification: 5th International Conference, RV 2014. Proceedings, Springer, 2014, pp. 92–107.

[5] M. Herlihy, Wait-free synchronization, ACM Trans. Program. Lang. Syst. 13 (1991) 124–149.

[6] M. Mostafa, B. Bonakdarpour, Decentralized runtime verification of LTL specifications in distributed systems, in: Parallel and Distributed Processing Symposium, IEEE International Conference, IPDPS 2015. Proceedings, 2015, pp. 494–503.

[7] Y. Falcone, T. Cornebize, J.-C. Fernandez, Efficient and generalized decentralized monitoring of regular languages, in: E. Ábrahám, C. Palamidessi (Eds.), Formal Techniques for Distributed Objects, Components, and Systems: 34th IFIP WG 6.1 International Conference, FORTE 2014. Proceedings, Springer, 2014, pp. 66–83.

[8] E. Bartocci, Sampling-based decentralized monitoring for networked embedded systems, in: HAS, volume 124 of *EPTCS*, 2013, pp. 85–99.

[9] M. Wooldridge, N. R. Jennings, Intelligent agents: theory and practice, Knowledge Eng. Review 10 (1995) 115–152.

[10] A. Ferrando, D. Ancona, V. Mascardi, Decentralizing MAS monitoring with decamon, in: K. Larson, M. Winikoff, S. Das, E. H. Durfee (Eds.), Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017, ACM, 2017, pp. 239–248. URL: http://dl.acm.org/citation.cfm?id=3091164.

[11] D. Ancona, D. Briola, A. Ferrando, V. Mascardi, MAS-DRiVe: a practical approach to decentralized runtime verification of agent interaction protocols, in: C. Santoro, F. Messina, M. D. Benedetti (Eds.), From Objects to Agents, 17th Workshop, WOA 2016. Proceedings, volume 1664 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 35–43. URL: http://ceur-ws.org/Vol-1664.

[12] D. Ancona, A. Ferrando, L. Franceschini, V. Mascardi, Coping with bad agent interaction protocols when monitoring partially observable multiagent systems, in: PAAMS, volume 10978 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 59–71.

[13] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, S. A. Smolka, Model repair for probabilistic systems, in: TACAS, volume 6605 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 326–340.

[14] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, E. Zadok, Runtime verification with state estimation, in: RV, volume 7186 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 193–207.

[15] Y. Joshi, G. M. Tchamgoue, S. Fischmeister, Runtime verification of LTL on lossy traces, in: SAC, ACM, 2017, pp. 1379–1386.

[16] R. Babaee, A. Gurfinkel, S. Fischmeister, $Prevent:$ A predictive run-time verification

framework using statistical learning, in: SEFM, volume 10886 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 205–220.

[17] N. Criado, J. M. Such, Norm monitoring under partial action observability, IEEE Trans. Cybernetics 47 (2017) 270–282.

[18] N. Criado Pacheco, Resource-bounded norm monitoring in multi-agent systems, Journal Artificial Intelligence Research (2018) 1.

[19] M. Baldoni, C. Baroglio, F. Capuzzimati, A commitment-based infrastructure for programming socio-technical systems, ACM Trans. Internet Techn. 14 (2014) 23:1–23:23. doi:10.1145/2677206.

[20] M. Baldoni, C. Baroglio, F. Capuzzimati, R. Micalizio, Exploiting social commitments in programming agent interaction, in: Proc. of PRIMA 2015, volume 9387 of *LNCS*, Springer, 2015, pp. 566–574.

[21] A. K. Chopra, M. P. Singh, Cupid: Commitments in relational algebra, in: Proc. of AAAI 2015, AAAI Press, 2015, pp. 2052–2059.

[22] M. Winikoff, W. Liu, J. Harland, Enhancing commitment machines, in: Proc. of DALT 2004, Revised Selected Papers, volume 3476 of *LNCS*, Springer, 2004, pp. 198–220.

[23] P. Yolum, M. P. Singh, Commitment machines, in: Proc. of ATAL 2001, Revised Papers, volume 2333, Springer, 2002, pp. 235–247.

[24] A. K. Chopra, S. Christie, M. P. Singh, Splee: A declarative information-based language for multiagent interaction protocols, in: Proc. of AAMAS 2017, ACM, 2017, pp. 1054–1063.

[25] M. P. Singh, Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language, in: Proc. of AAMAS 2011, IFAAMAS, 2011, pp. 491–498.

[26] M. Winikoff, N. Yadav, L. Padgham, A new hierarchical agent protocol notation, Autonomous Agents and Multi-Agent Systems 32 (2018) 59–133.

[27] D. Ancona, A. Ferrando, V. Mascardi, Improving flexibility and dependability of remote patient monitoring with agent-oriented approaches, Int. J. Agent Oriented Softw. Eng. 6 (2018) 402–442. URL: https://doi.org/10.1504/IJAOSE.2018.096422. doi:10.1504/IJAOSE.2018.096422.

[28] D. Ancona, A. Ferrando, V. Mascardi, Mind the gap! Runtime verification of partially observable MASs with probabilistic trace expressions, in: D. Baumeister, J. Rothe (Eds.), The 19th European Conference on Multi-Agent Systems (EUMAS 2022), Springer, 2022.

[29] D. Ancona, S. Drossopoulou, V. Mascardi, Automatic generation of self-monitoring MASs from multiparty global session types in Jason, in: Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers, 2012, pp. 76–95.

[30] D. Ancona, M. Barbieri, V. Mascardi, Constrained global types for dynamic checking of protocol conformance in multi-agent systems, in: SAC, ACM, 2013, pp. 1377–1379.

[31] D. Briola, V. Mascardi, D. Ancona, Distributed runtime verification of JADE multiagent systems, in: IDC, volume 570 of *Studies in Computational Intelligence*, Springer, 2014, pp. 81–91.

[32] D. Ancona, D. Briola, A. Ferrando, V. Mascardi, Global protocols as first class entities for self-adaptive agents, in: AAMAS, ACM, 2015, pp. 1019–1029.

[33] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. M. et al., Behavioral types in programming languages, Foundations and Trends in Programming Languages 3 (2016) 95–230.

[34] D. Ancona, A. Ferrando, V. Mascardi, Comparing trace expressions and linear temporal logic for runtime verification, in: Theory and Practice of Formal Methods, volume 9660 of *LNCS*, 2016, pp. 47–64.

[35] D. Ancona, A. Ferrando, V. Mascardi, Parametric runtime verification of multiagent systems, in: AAMAS, ACM, 2017, pp. 1457–1459.

[36] D. Ancona, A. Ferrando, L. Franceschini, V. Mascardi, Parametric trace expressions for runtime verification of Java-like programs, in: FTfJP@ECOOP, ACM, 2017, pp. 10:1–10:6.

[37] D. Ancona, L. Franceschini, A. Ferrando, V. Mascardi, RML: theory and practice of a domain specific language for runtime verification, Sci. Comput. Program. 205 (2021) 102610. URL: https://doi.org/10.1016/j.scico.2021.102610. doi:10.1016/j.scico.2021.102610.

[38] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (1978) 558–565. URL: http://doi.acm.org/10.1145/359545.359563. doi:10.1145/359545.359563.

[39] D. Briola, V. Mascardi, D. Ancona, Distributed runtime verification of JADE and Jason multiagent systems with Prolog, in: CILC, volume 1195 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014, pp. 319–323.

[40] A. Ferrando, R. C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, V. Mascardi, Rosmonitoring: A runtime verification framework for ROS, in: A. Mohammad, X. Dong, M. Russo (Eds.), Towards Autonomous Robotic Systems - 21st Annual Conference, TAROS 2020, Nottingham, UK, September 16, 2020, Proceedings, volume 12228 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 387–399. URL: https://doi.org/10.1007/978-3-030-63486-5_40. doi:10.1007/978-3-030-63486-5_40.

[41] M. Leotta, D. Ancona, L. Franceschini, D. Olianas, M. Ribaudo, F. Ricca, Towards a runtime verification approach for internet of things systems, in: C. Pautasso, F. Sánchez-Figueroa, K. Systä, J. M. M. Rodriguez (Eds.), Current Trends in Web Engineering - ICWE 2018 International Workshops, MATWEP, EnWot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised Selected Papers, volume 11153 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 83–96. URL: https://doi.org/10.1007/978-3-030-03056-8_8. doi:10.1007/978-3-030-03056-8_8.