Decentralizing MAS Monitoring with DecAMon

Angelo Ferrando * Davide Ancona Viviana Mascardi
Dept. of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS)
University of Genova, Italy
angelo.ferrando@dibris.unige.it, davide.ancona@unige.it, viviana.mascardi@unige.it

ABSTRACT
We describe DecAMon, an algorithm for decentralizing the monitoring of the MAS communicative behavior described via an Agent Interaction Protocol (AIP). If some agents in the MAS are grouped together and monitored by the same monitor, instead of individually, a partial decentralization of the monitoring activity can still be obtained even if the “unique point of choice” (a.k.a. local choice) and “connect- edness for sequence” (a.k.a. causality) coherence conditions are not satisfied by the protocol. Given an AIP specification, DecAMon outputs a set of “Monitoring Safe Partitions” of the agents, namely partitions $P$ which ensure that having one monitor in charge for each group of agents in $P$ allows detection of all and only the protocol violations that a fully centralized monitor would detect. In order to specify AIPs we use “trace expressions”: this formalism can express event traces that are not context-free and can model both synchronous and asynchronous communication just by changing the underlying notion of event.

CCS Concepts
- Computing methodologies → Multi-agent systems;

Keywords
Decentralized Runtime Verification, MAS Runtime Verification, Agent Interaction Protocol

1. INTRODUCTION
Ensuring that software applications behave as expected is a challenging task in general, and it is even more demanding when the software application is as large, distributed, and communication-intensive as a MAS.

By complementing formal static verification and testing, Runtime Verification (RV) offers a practical solution for detecting some misbehavior which can emerge at runtime: in RV, a monitor is generated from a formal specification of the properties to be verified and dynamically checks the behavior of the monitored system w.r.t. the given specification. When the system is small, one centralized monitor can check it all without becoming a bottleneck. Nevertheless, centralized monitoring does not scale with the growth of the system dimension and a decentralized monitoring approach may be the only viable solution for coping with the system complexity. Also, decentralized monitoring may be a more natural choice when the system is distributed, since different monitors can be associated with groups of physically, geographically, or logically connected entities, gaining in efficiency and modularity.

If we limit ourselves to the MAS communicative behavior, we can assume that agents interactions can be observed by adding unobtrusive sniffers to the MAS communication layer, as happens for example in JADE [11] and as can be obtained with limited effort in Jason [13].

Associating an individual agent or group of agents with a sniffer in charge of observing their communicative behavior does not raise serious technical problems if JADE or Jason are used. Rather, the actual problems for dynamically verifying that a MAS behaves according to a given Agent Interaction Protocol (AIP) are:

1. how to specify the expected global communicative behavior of the MAS (the global AIP) in a formal way,
2. how to automatically derive the partial AIPs associated with each agent or group of agents from the global one,
3. how to turn a sniffer into a monitor that, besides observing agent interactions, is able to check their compliance w.r.t. to the AIP formal specification and, above all,
4. how to ensure that the system made up of the decentralized monitors detects all and only the same protocol violations that a single centralized monitor observing the MAS would detect.

Solutions to the first three problems exist and will be briefly discussed in the paper, but the last issue is still open and extremely challenging: it is indeed well known that some global AIPs cannot be fully decentralized for monitoring purposes (namely, they cannot be properly monitored by individual monitors associated with individual agents), because they do not respect some necessary coherence conditions [29, 30]. In the existing literature these protocols are either monitored in a centralized way or discarded.

In this paper we describe DecAMon, an algorithm for Decentralizing the Agent system Monitoring which works also in case the global AIP specification, expressed using trace expressions [8], does not satisfy those coherence conditions. Trace expressions represent a suitable solution to problem 1, “how to specify the MAS communicative behavior in a formal way”. 
Our proposal, which falls in the Decentralized Runtime Verification area [12], is based on the idea that, between a fully centralized and a fully decentralized monitoring approach, a third viable solution is possible: partially decentralized monitoring.

DecAMon is completely agnostic w.r.t. the type of observed events. Although in this paper, for sake of presentation clarity, we limit ourselves to consider interaction events, the DecAMon algorithm can be applied to trace expressions dealing with events of any kind. The only requirement for the DecAMon algorithm to work is that the set of agents involved in a given event should be efficiently computed.

The paper is organized in the following way: Section 2 discusses the motivations of our work; Section 3 gently presents the DecAMon algorithm to work is that the set of agents involved in a given event should be efficiently computed.

Alice and Carol are two PhD students. They are writing a paper for the AAMAS 2017 conference with their colleague Bob and their supervisor Dave. Alice and Carol are in charge for running experiments. They are working on a shared repository and they agree on the notion of satisfactory results. A few weeks before the deadline, they decide that if the experimental results will reach a satisfactory level, they will make a first submission in the morning and then she will submit the final version to their supervisor.

The first problem is that these protocols are too abstract. Although they make sense in formal computer science, they do not clearly state what the behavior of each agent is or how the experiments should be run. This can lead to confusion among readers who are not familiar with the subject.

The second problem is that these protocols are too abstract. They do not clearly state what the behavior of each agent is or how the experiments should be run. This can lead to confusion among readers who are not familiar with the subject.

The problem with AIP\(_1\) is that involves all the agents in the MAS to a view of that protocol restricted to an agent subset is usually named “projection”. If \(\mathcal{T}\) identifies the set of trace expressions, projection can be described as a function \(\Pi : \mathcal{T} \times \mathcal{P}(\text{Ags}) \rightarrow \mathcal{T}\), where the second argument is the subset of agents onto which projection is made. Trace expression projection can be formally defined and effectively computed [3, 6] and provides a solution to the second problem introduced in Section 1: “how to automatically derive the partial AIPs associated with each individual agent or group of agents in the MAS from the global one”. Even if we are able to project onto individual agents, how can we be sure that the individual monitoring gives the same results as the centralized one? This second problem is independent from the granularity of communicative events. Rather, it depends on taking a centralized or a decentralized view point. Let us consider AIP\(_1\). A monitor \(M_{\text{alice}}\) associated with Alice and driven by the protocol portion that involves Alice only, will consider her behavior correct if she sends a meet message to Dave. In the same way, a monitor \(M_{\text{carol}}\) will consider Carol’s behavior correct if she sends a meet message to Bob. However, performing both actions will not be compliant with AIP\(_1\) as they are mutually exclusive. Unfortunately, none among \(M_{\text{alice}}, M_{\text{carol}}, M_{\text{bob}}, M_{\text{dave}}\) alone can verify if mutual exclusivity is respected. Following the terminology from the behavioral types literature, AIP\(_1\) does not satisfy the unique point of choice coherence condition; using the terminology from message flow graphs theory, it shows a non local choice problem [28].

The problem with AIP\(_2\) is different: what happens if Carol sends submitted to Dave before Alice submits the paper? The AIP\(_2\) portion that \(M_{\text{carol}}\) sees is

\[ \text{carol submitted} \rightarrow \text{dave : } \epsilon \]

We name these protocols CAIP\(_1\) and CAIP\(_2\) to stress that they are “Concrete”. AIP\(_2\) = \(a\langle m_b \rangle::(a\langle m_b \rangle::(a\langle c \rangle::(m_d::c::\epsilon)\epsilon)\epsilon)\epsilon\)

The AIP\(_2\) protocol allows for a more flexible approach to monitoring, as it allows for the possibility of multiple agents communicating simultaneously. This can lead to situations where multiple agents can communicate with each other, and the monitoring system must be able to handle this complexity.

By measuring both a synchronous behavior, where nothing should happen between a sending and the corresponding reception (we use a for Alice, b for Bob, and c for Carol), AIP\(_1\) = \(a\langle m_b \rangle::(a\langle m_b \rangle::(a\langle c \rangle::(m_d::c::\epsilon)\epsilon)\epsilon)\epsilon\).

The shuffle operator combines two protocols by allowing any shuffling of the events at its right with those at its left. Of course, ordering among events in each branch must be preserved. The idea behind \(\tau_1|\tau_2\) is that \(\tau_1\) and \(\tau_2\) are independent.

The AIP\(_2\) protocol allows for a more flexible approach to monitoring, as it allows for the possibility of multiple agents communicating simultaneously. This can lead to situations where multiple agents can communicate with each other, and the monitoring system must be able to handle this complexity.
On the other hand, the portion seen by $M_{\text{alice}}$ is

$$\text{alice} \xrightarrow{\text{submit}} \text{aamas} : (\text{alice} \xrightarrow{\text{submit}} \text{aamas} : \epsilon \lor \epsilon)$$

No individual monitor can state whether a submitted message sent by Carol to Dave comes after Alice completed her last submission. According to the behavioral types taxonomy, AIP$_2$ does not satisfy the connectedness for sequence coherence condition. According to Desai and Singh’s terminology [21], AIP$_2$ problem is due to the blindness of Carol w.r.t. the submit message that Alice sends to aamas.

A protocol that meets the coherence conditions can always be fully decentralized as protocol violations are only due to messages which are exchanged in a given protocol state, but were not allowed in that state. For example, AIP$_1 = (\text{submit} aamas \xrightarrow{\text{submit}} \text{dave} : \epsilon \lor \text{aamas} \xrightarrow{\text{ack}} \text{alice} : \epsilon)$ could be violated by Alice submitting the paper twice. However, the second $\text{alice} \xrightarrow{\text{submit}} \text{aamas}$ interaction would violate both $\Pi(AIP_1, \{\text{alice}\})$ and $\Pi(AIP_2, \{\text{aamas}\})$ (the projections of the global protocol AIP$_2$ onto $\{\text{alice}\}$ and $\{\text{aamas}\}$, respectively), so at least one decentralized monitor between $M_{\text{alice}}$ and $M_{\text{aamas}}$ would immediately detect it. A protocol that does not meet the coherence conditions causes problems only when we try to fully decentralize its monitoring: each agent $ag$ has its own monitor that checks if $ag$ behavior is compliant with $\Pi(\tau, \{ag\})$. This may cause the loss of sequentiality and mutual exclusivity constraints. As long as we assume that centralized monitoring takes place no problems arise, apart from the enormous bottleneck that the centralized monitor may become!

Given a protocol specification and the set $\text{Ags}$ of agents in the MAS, DecAMon faces the partial decentralization problem by computing a set of “Monitoring Safe (MS) partitions” of $\text{Ags}$. If a violation of the behavior patterns defined by the protocol takes place, one monitor in charge for one group in the MS partition will detect it.

3. **DECAMON: A GENTLE INTRODUCTION**

Let us suppose that the agents involved in the MAS are $\text{alice}$, $\text{bob}$, $\text{carol}$, and $\text{dave}$. If $\{ \{\text{alice}, \text{carol}\}, \{\text{bob}\}, \{\text{dave}\} \}$ is a MS partition, then $\text{alice}$ and $\text{carol}$ must be monitored by the same monitor $M_{\{\text{alice}, \text{carol}\}}$, whereas $\text{bob}$ and $\text{dave}$ may be monitored by distinct monitors. This does not mean that having one monitor $M_{\{\text{alice}, \text{carol}\}}$ for $\text{alice}$ and $\text{carol}$ and one $M_{\{\text{bob}, \text{dave}\}}$ for $\text{bob}$ and $\text{dave}$ (to be monitored together), or one single monitor $M_{\{\text{alice}, \text{bob}, \text{carol}, \text{dave}\}}$ for all the four agents, is not monitoring safe: larger groups can be formed, provided that those agents which must stay together, are monitored together. The above partition is one of those returned by DecAMon on the AIP$_1$ protocol introduced in Section 2: if the same monitor observes both $\text{alice}$ and $\text{carol}$, it will be able to detect violations of mutual exclusivity between $\text{alice} \xrightarrow{\text{submit}} \text{dave}$ and $\text{carol} \xrightarrow{\text{submit}} \text{bob}$.

In a similar way, one MS partition of the agents involved in AIP$_2$ is $\{ \{\text{alice}, \text{dave}\}, \{\text{aamas}\}, \{\text{carol}\}\}$: if the same monitor is in charge for both $\text{alice}$ and $\text{dave}$, it can verify that the interaction involving $\text{dave}$ (and $\text{carol}$) takes place after the interactions involving $\text{alice}$ (and $\text{aamas}$).

**Intuition 3.1 (Monitoring Safety (MS)).** A partition of $\text{Ags}$ $P$ is Monitoring Safe (MS partition, abbreviated in MS in the sequel) if it enjoys the following property: if the agents belonging to the same group in $P$ are monitored together, no loss of sequentiality and mutual exclusivity constraints takes place; one among the decentralized monitors detects a violation of “its portion” of the global protocol iff a violation of the global protocol occurs.

If the system monitoring cannot be decentralized, DecAMon will return only one MS, $\{\text{Ags}\}$. On the other hand, if each agent $ag_i \in \text{Ags}$, with $i \in \{1, \ldots, n\}$ can be monitored independently from the others, DecAMon will output $\{\{ag_1\}, \{ag_2\}, \{ag_3\}, \ldots, \{ag_n\}\}$.

DecAMon isomorphism w.r.t. the events syntax gives us the flexibility to execute it on abstract and concrete agent protocol specifications by defining the involved function as

$$\text{involved}(ag_1 \xrightarrow{msg} ag_2) = \{ag_1, ag_2\}$$
$$\text{involved}(ag_1 \xrightarrow{msg \oplus ag_2}) = \{ag_1\}$$
$$\text{involved}(ag_1 \xrightarrow{msg \oplus ag_2}) = \{ag_2\}$$

When we adopt a concrete protocol perspective, where sending and reception are distinct, the only entity involved in a message sending (resp. reception) is the sender (resp. the receiver), even if we keep track of the message sender also in a “receive” event $(ag_1 \xrightarrow{msg \oplus ag_2} ag_2)$ and vice versa.

When we adopt a concrete protocol perspective, where sending and reception are distinct, the only entity involved in a message sending (resp. reception) is the sender (resp. the receiver), even if we keep track of the message sender also in a “receive” event $(ag_1 \xrightarrow{msg \oplus ag_2} ag_2)$ and vice versa.

Continuing the AIP$_1$ example, the other MSs are $\{\{\text{alice}\}, \{\text{carol}\}, \{\text{bob}, \text{dave}\}\}$, $\{\{\text{alice}, \text{bob}\}, \{\text{carol}\}, \{\text{dave}\}\}$, $\{\{\text{alice}\}, \{\text{carol}, \text{dave}\}, \{\text{bob}\}\}$.

If $ev_1$ and $ev_2$ are joined by a $\lor$ operator in the AIP like $\text{alice} \xrightarrow{msg \lor dave} \text{carol} \xrightarrow{msg \lor \text{bob}}$, then $ev_1 \lor ev_2$ has empty intersection with $\text{involved}(ev_1)$, one agent $ag_1 \in \text{involved}(ev_1)$ must be monitored together with one agent $ag_2 \in \text{involved}(ev_2)$ to ensure that mutual exclusivity becomes verifiable. In a similar way, if $ev_1$ and $ev_2$ are two sequential events like $\text{alice} \xrightarrow{\text{submit}} \text{aamas}$ and $\text{carol} \xrightarrow{\text{submit}} \text{dave}$ in AIP$_2$, and $\text{involved}(ev_1)$ has empty intersection with $\text{involved}(ev_2)$, one agent $ag_1 \in \text{involved}(ev_1)$ must be monitored together with one agent $ag_2 \in \text{involved}(ev_2)$ in order to verify the correct sequentiality of $ev_1$ and $ev_2$. In both cases, if there exists one agent $agI \in \text{involved}(ev_1) \cap \text{involved}(ev_2)$ no grouping is required: the monitor associated with $agI$ can verify mutual exclusiveness and correct sequencing between $ev_1$ and $ev_2$.

3.1 **Trace expressions**

This subsection is based on [8]. We simplified some definitions for sake of presentation.

Trace expressions are a formalism expressly designed for RV. In the following we denote by $E$ a fixed universe of events $(\text{carol} \xrightarrow{\text{submit}} \text{dave}$ is an event) and by $\mathcal{E}$ a fixed universe of event types each denoting a subset of events in $E$ (SubmitType $= \{\text{alice} \xrightarrow{\text{submit}} \text{aamas}, \text{carol} \xrightarrow{\text{submit}} \text{dave}\}$ is an event type). Since any trace expression built on top of event types containing finitely many events can be automatically transformed into an equivalent trace expression built on top of singleton event types (event types containing exactly one event), we will assume that DecAMon operates on trace expressions with singleton event types only.

**Event traces.** An event trace over $E$ is a possibly infinite sequence of events in $E$. A trace expression over $\mathcal{E}$ denotes a set of event traces over $\mathcal{E}$. In other words, the denotational semantics of a trace expression is a set of event traces.

²W.r.t. trace expression denotational semantics.
Trace expressions. A trace expression τ is defined on top of the following operators; binary operators associate from left to right and the topmost one has the highest precedence.

- \(\epsilon\) (empty trace), denoting the singleton set \{\(\epsilon\)\} containing the empty event trace \(\epsilon\).
- \(\emptyset \tau\) (prefix), denoting the set of all traces whose first event \(ev\) matches the event type \(\emptyset\) (\(ev \in \emptyset\)), and the remaining part is a trace of \(\tau\).
- \(\tau_1 \tau_2\) (concatenation), denoting the set of all traces obtained by concatenating the traces of \(\tau_1\) with those of \(\tau_2\).
- \(\tau_1 \wedge \tau_2\) (intersection), denoting the intersection of the traces of \(\tau_1\) and \(\tau_2\).
- \(\tau_1 \vee \tau_2\) (union), denoting the union of the traces of \(\tau_1\) and \(\tau_2\).
- \(\tau_1 \lhd \tau_2\) (shuffle), denoting the set obtained by shuffling the traces of \(\tau_1\) with the traces of \(\tau_2\).

With some abuse of notation, in the sequel we will use events instead of event types inside trace expressions. This is the notation we already used in Section 2, where protocols were built on top of events.

To support recursion without introducing an explicit construct, trace expressions are cyclic terms which can be represented by a finite set of syntactic equations, as happens in Jason and in most modern Prolog implementations. To make an example, AIP\(_4\) represented by \(\tau = \text{alice} : \text{submit}\) aamas : (\(\tau \lor \epsilon\)) · carol : \(\\text{submitted}\) dave : \(\epsilon\) models the protocol where alice can submit a paper to aamas one or more (including infinitely many) times, before carol sends a submitted message to dave. If we name \(a\) the event alice : submit aamas and \(c\) the event carol : submit dave, the denotational semantics of \(\tau\) is the set of event traces \((ac, aac, aaac, \ldots, a^n c, a^n)\) where \(a^n\) stands for a trace containing \(n\) instances of the event alice : submit aamas and \(a^0\) stands for an infinite trace of alice : submit aamas events.

### 3.2 High-level Description and Examples

Now, we are ready to introduce the notions of critical point of a trace expression and of minimality of a MS partition. The function \(\text{first}(\tau)\) returns all the first events of \(\tau\) and \(\text{last}(\tau)\) all its last events. For example, \(\text{first}(a \lor b : c : \epsilon) = \{a, b\}\) and \(\text{last}(a \lor b : c : \epsilon) = \{a, c\}\). Their precise definition is given in Section 4.

**Def. 3.1 (Critical Point).** A couple of events \((ev_1, ev_2)\) is a critical point of \(\tau\) iff \(\tau_{\text{sub}}\) is a sub-expression of \(\tau\) such that

- \(\tau_{\text{sub}} = \tau_1 \wedge \tau_2\) and \(\exists ev_1 \in \text{first}(\tau_1), \exists ev_2 \in \text{first}(\tau_2)\) s.t. involved\((ev_1) \cap \text{involved}(ev_2) = \emptyset\), or
- \(\tau_{\text{sub}} = ev_1 \tau_2\) and \(\exists ev_2 \in \text{first}(\tau_2)\) s.t. involved\((ev_1) \cap \text{involved}(ev_2) = \emptyset\), or
- \(\tau_{\text{sub}} = \tau_1 \tau_2\) and \(\exists ev_1 \in \text{last}(\tau_1), \exists ev_2 \in \text{first}(\tau_2)\) s.t. involved\((ev_1) \cap \text{involved}(ev_2) = \emptyset\).

We say that \(\tau_{\text{sub}}\) generates the critical point \((ev_1, ev_2)\). Uniqueness of \((ev_1, ev_2)\) is violated if both \(ev_1\) and \(ev_2\) take place. Sequentiality of \((ev_1, ev_2)\) is violated if \(ev_2\) takes place before \(ev_1\).

**Def. 3.2 (Minimal Monitoring Safety (MMS)).** The partition \(P\) of agents \(\text{Ags}\) is Minimal Monitoring Safe (MMS) if it is Monitoring Safe and if splitting one of the groups of agents in \(P\) leads to a partition that does not satisfy monitoring safety any longer.

For generating a set of MSs, DecAMon exploits merge: merge \(\in \mathcal{P}^{\text{P}(\text{Ags})} \times \mathcal{P}^{\text{P}(\text{Ags})} \rightarrow \mathcal{P}^{\mathcal{P}(\text{Ags})}\).

One argument \(C\) of merge (no matter which, since merge is commutative) consists of new groups of agents that are constrained to be monitored together; the other argument \(\text{OldC}\) models the existing agent grouping constraints: we name them “constraint stores”. The result of merge is a new constraint store \(\text{NewC}\) where both the constraints in \(\text{OldC}\) and those in \(C\) hold. No unnecessary constraints (namely, no unnecessary groupings) are added to the merge result. The way merge works ensures that the groups of agents in \(\text{NewC} \in \mathcal{P}(\text{Ags})\) will be disjoint if the groups of agents in \(C\) were disjoints, and the groups of agents in \(\text{OldC}\) were. Let us introduce merge by means of an example: the idea behind merge\(\{\{\text{ag1, ag2}\}\}, \{\{\text{ag1, ag3}\}, \{\text{ag4, ag5}\}\}\) is to add the new constraint “agents ag1 and ag2 must be monitored together” to the constraint store \(\{\{\text{ag1, ag3}\}, \{\text{ag4, ag5}\}\}\), stating that ag1 and ag3 must be monitored together, as well as ag4 and ag5. The only constraint store resulting from this merge is \(\text{NewC} = \{\{\text{ag1, ag2, ag3}\}, \{\text{ag4, ag5}\}\}\), where both the previous and the new constraints are respected. The amount of agents that are constrained to be monitored together is minimized: in \(\text{NewC}\), ag1 and ag4 can be monitored independently as there is no reason to group them. The constraint store \(\text{NewC} = \{\{\text{ag1, ag2, ag3, ag4, ag5}\}\}\) satisfies the old and new constraints as well but uselessly imposes that ag1 and ag4 are monitored together: merge will never return it.

DecAMon carries out a structural analysis of the trace expression in order to find those agents that must be monitored together because they are involved in a critical point. As soon as new groups of agents that must be monitored together are found, the new constraint store is merged with the previously computed one: given a trace expression \(\tau = \tau_1 \op \tau_2\) where \(\op\) is a binary operator, DecAMon computes the constraint stores due to \(\op\) (they may be more than one, as shown in the sequel) and computes the combinations obtained by merging each of them with each of those resulting from \(\tau_1\) and each of those resulting from \(\tau_2\). Since the constraint stores deriving from \(\tau_1\) and \(\tau_2\) are computed independently, they could overlap up to some extent and their merge could generate groupings with unnecessary constraints.

To cope with this problem we have implemented a post-processing algorithm that allows us to obtain the set of MMS constraints as a refinement of the DecAMon output, by removing those MMSs that add useless constraints to other MMSs. The global minimality property can be obtained either via this post-processing activity where each returned MMS is compared with all the others, or by making merge more complex (merge would need to know all the possible MMSs for each trace expression branch to discard the overlapping ones and return only minimal MMSs). We opted for the first solution.

Let us consider another example: if we had to compute merge\(\{\{\text{ag1, ag2, ag4}\}\}, \{\{\text{ag1, ag3}\}, \{\text{ag2, ag5}\}\}\) the only possible result would be to merge \(\{\text{ag1, ag3}\}\) and \(\{\text{ag2, ag5}\}\) where ag1 and ag2 could be monitored independently, to meet the new constraint where they must be monitored together; a4 must be grouped with a1 and a2. The result is \(\{\{\text{ag1, ag2, ag3, ag4, ag5}\}\}\).
as \((ab|c|e|d|f|g|h|i|x)\) · \((jk|e|lm|x|no)\) where \(ab\) stands for \(\frac{ab}{a},\ b\) stands for \(\frac{b}{b}\), and so on.

DecAMon starts exploring AIP\(_2\) looking for critical points. The outmost AIP\(_2\) operator is a concatenation, which may generate critical points. DecAMon computes all the last events in \(\tau_1 = \{abcdc\} \cdot \{def\} \cdot \{gh\} \cdot \{i\} \cdot \{j\} \cdot \{k\} \cdot \{l\} \cdot \{m\} \cdot \{n\} \cdot \{o\}\) and all the first events in \(\tau_2 = \{jk\} \cdot \{lm\} \cdot \{no\}\). They turn out to be \(last(\tau_1) = \{bc, ef, hi\}\) and \(first(\tau_2) = \{jk, lmn, no\}\).

Any couple \(ev_1, ev_2\) s.t. \(ev_1 \in last(\tau_1), ev_2 \in first(\tau_2), and involved(ev_1) \cap involved(ev_2) = \emptyset\) is a critical point. Here, \(bc, ef, hi\), \(ef, lm, no\), \(hi, jk\), \(hi, lm\), \(hi, no\), are all the critical points generated by the outmost \(\cdot\) in AIP\(_2\).

For each critical point \((ev_1, ev_2)\), one agent involved in \(ev_1\) must be grouped together with one agent involved in \(ev_2\).

**Def. 3.3 (Critical Point Satisfaction).** A group of agents satisfies a critical point \((ev_1, ev_2)\) if it contains one agent involved in \(ev_1\) and one agent involved in \(ev_2\).

**Def. 3.4 (Trace Expression Satisfaction).** A constraint store \(C\) satisfies a trace expression if all the critical points generated by the outmost operator in that trace expression are satisfied by one group in \(C\).

To make another example, \(C_{51} = \{b, c, e, h, j, l, n\}\) satisfies AIP\(_2\) \(\tau_1 = \tau_2\) since \(b\) which is involved in \(bc\) is grouped with \(j\) involved in \(jk\), \(l\) involved in \(lm\), and \(n\) involved in \(no\). The same holds for \(e\) involved in \(ef\) and \(h\) involved in \(hi\).

Also \(C_{52} = \{b, c, e, f, h, j, l, n\}\) satisfies AIP\(_2\) \(\tau_1 = \tau_2\) and AIP\(_2\) \(\tau_1\) is grouped with \(k, m, o\) hence satisfying \(bc, jk, ef, lm\), and \(bc, no\); \(e\) and \(h\) are grouped with \(j, l, n\), hence satisfying \(ef, jk\), \(ef, lm\), \(ef, no\), \(hi, jk\), \(hi, lm\), and \(hi, no\).

The same holds for \(C_{53} = \{c, k, c, e, f, h, j, l, n\}\): \(c, k\) satisfies \(bc, jk\); \(b, m, o\) satisfies \(bc, ef\) and \(bc, no\); \(e, h, j, l, n\) satisfies all the remaining critical points.

The constraint store \(\{(c, j), (f, l), (i, n), \{a\}, \{b\}, \{d\}, \{e\}, \{g\}, \{h\}, \{k\}, \{m\}, \{o\}\}\) instead, does not satisfy AIP\(_2\): for example, no group satisfies \(bc, ef, lm\).

We define \(C_{70}\) as the constraint store that contains all and only one singleton set \(\{ag\}\) for each agent \(ag\) involved in \(\tau\). Given the initial constraint store \(C_{50}\) for the protocol AIP\(_2\), DecAMon merges \(C_{50}\) with one of the constraint stores \(C_{5i}\) that satisfy AIP\(_2\), selected on a nondeterministic basis. Then, it recursively explores the components \(\tau_1\) and \(\tau_2\) of AIP\(_2\) and adds the newly discovered constraints to the previously computed constraint store. The sequences \(abcdc\), \(def\), and \(gh\) \(i\) in \(\tau_1\) do not generate any new critical point because they verify the connectedness for sequence condition. Moreover, they are joined by a shuffle operator that generates no critical points. Thus, no new constraints are generated because of \(\tau_1\). In a similar way, no new constraints are generated because of \(\tau_2\).

The nondeterministic selection of one of the constraint stores satisfying the currently analyzed trace expression is repeated for each possible constraint stores. By backtracking to any point of choice, DecAMon can produce all the possible MSs, one at a time: \(C_{51}, C_{52}, C_{53}, C_{54}\), together with other 5628 possible initial constraint stores, are the MSs output by DecAMon.

If \(\tau_1 = (abcdc) \cdot (efghxc)\) (AIP\(_2\)) the new constraints \(\{(a, c), (e, g)\}\) for \(\{(a, d), (f, g)\}\) due to unique point of choice violation should have been merged with \(C_{5i}\), giving a different (and smaller) final set of MSs.

If \(\tau_1 = (abcdc) \lor (efghxc)\) (AIP\(_2\)) a further constraint \(\{a, e\}\) (or \(\{a, f\}\), or \(\{b, e\}\), or \(\{b, f\}\) due to unique point of choice violation should have been merged with the previous ones.

Finally, let us consider the concrete protocol CAIP\(_1\) = \(a_{\{m, b\}}: b = a_{\{m, c\}}: c = \cdot\). As anticipated, DecAMon works exactly in the same way, provided it can compute the involved function. CAIP\(_1\) can be seen as \(a_{\{m, b\}}: \tau_1\). Its outmost operator is the first prefix with \(a_{\{m, b\}}: \tau_1\) at its left, \(first(\tau_1) = \{a, b\}\). These two events share no involved agents: \(a_{\{m, b\}}: \tau_1\) is a critical point. The constraint store generated by CAIP\(_1\) is \(\{\{a, b\}\}\) which must be merged with the initial constraint store \(\{\{a\}, \{b\}, \{c\}\}\) leading to \(\{\{a, b\}\}, \{\{b\}, \{c\}\}\). Now DecAMon is called on \(\tau_1 = \{\{a, b\}\}: \{\{a, c\}\} = \{\{a, b\}\} = \{\{a, c\}\}\) generating the new constraint store \(\{\{a, c\}\}\) which must be merged with \(\{\{a, b\}\}, \{\{b\}, \{c\}\}\). In the end, all the three agents must be monitored together. This is correct: how can \(M_b\) alone verify that when \(b\) receives \(m_1\) from \(a\), \(a\) actually sent it before? If we make either security assumptions ("all the received messages have been actually sent by the sender") or strong assumptions on the underlying network reliability ("all the sent messages will be received", or even "all the sent messages will be received in the same order they were sent") we can relax some monitoring safety constraints, but this is not possible in general.

**4. DESIGN, IMPLEMENTATION, EXPERIMENTS**

In order to provide a formal account of runtime verification using trace expressions, it is useful to explain what we mean by "trace expression transition relation". A trace expression can be seen as the state of an AIP. The notion of "transition from one trace expression (state of the protocol) to another" is at the basis of the trace expression operational semantics and makes AIP runtime verification possible. An event \(ev\) is compliant with the protocol current state iff the protocol can move to another state once \(ev\) has been observed.

The operational semantics of trace expressions is specified by the transition relation \(\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}\). \(\tau_1 \xrightarrow{\delta} \tau_2\) means \((\tau_1, ev, \tau_2) \in \delta\). If the trace expression \(\tau_1\) specifies the current valid state of the protocol, then an event \(ev\) is considered valid in the current state iff there exists a transition \(\tau_1 \xrightarrow{\delta} \tau_2\); in such a case, \(\tau_2\) specifies the next valid state of the protocol after event \(ev\) takes place. Otherwise, the event \(ev\) is not valid in \(\tau_1\).

Union, shuffle, and concatenation operators may lead to nondeterminism because for each of them two possibly overlapping transition rules are defined. We only consider deterministic and contractive trace expressions, where deterministic means that given a trace expression and an event, only one transition rule can be applied to them, and contractive means that all the infinite paths in the syntax tree corresponding to a trace expression contain the prefix operator. These restrictions do not limit trace expressions expressive power: the paper [8] demonstrates that deterministic and contractive trace expressions are more expressive than the three valued Linear Time Temporal Logic LTL\(_{3}\) [10].
the non empty traces of a trace expression, the predicate \( e(\cdot) \), inductively defined by the rules in Figure 2, defines the trace expressions that contain the empty trace \( \epsilon \) and hence may terminate. If \( e(\tau) \) holds, then the empty trace is a valid trace for \( \tau \). Both figures are taken from [8].

A SWI-Prolog implementation of these rules can be downloaded from \texttt{http://decamon.altervista.org/}; in the past, similar rules implementing the operational semantics of “global types” [7] have been integrated both in Jason, which supports a Prolog-like reasoning engine [4, 7], and in JADE, by means of the JPL Java–SWI-Prolog bidirectional interface, \texttt{http://www.swi-prolog.org/packages/jpl/java_api/} [14, 15]. This integration, which in the most recent works also supported projection, answers the third challenge raised in Section 1: “how to turn a sniffer into a monitor that, besides observing agent interactions, is able to check their compliance w.r.t. the AIP formal specification”. Although the results of the above papers make the decentralized runtime verification of Jason and JADE MASs possible from a practical point of view, they do not solve the theoretical problem stated in Section 1 about formal guarantees given by the decentralized monitoring process w.r.t. the centralized one.

### 4.1 Design

The definition of \( \text{first} \) does not need to take cycles – which are due to trace expressions recursive definitions – into account; in fact, contractions ensures that, while exploring a trace expression following its syntactical structure, a prefix operator will be met in a finite number of steps.

- \( \text{first}(\epsilon) = \{ \} \)
- \( \text{first}(\emptyset ; \tau) = \{ \emptyset \} \)
- \( \text{first}(\tau_1 \cdot \tau_2) = \text{first}(\tau_1) \cup \text{first}(\tau_2) \) if \( e(\tau_1) \);
- \( \text{first}(\tau_1 \cdot \tau_2) = \text{first}(\tau_1) \) otherwise
- \( \text{first}(\tau_1 \lor \tau_2) = \text{first}(\tau_1 \lor \tau_2) = \text{first}(\tau_1) \lor \text{first}(\tau_2) \).

The definition of \( \text{last} \) is more complex because it must not recall itself in case of cyclic trace expressions and contractions is not enough to avoid entering a loop. For example, \( \tau = \emptyset ; \cdot ; \tau \) is contractive, but we must have plenty of time and patience if we are going to look for its last element! In this case, \( \text{last} \) should return \( \{ \} \) (and we should do the same...) but it can do this only if it keeps track of the already met trace expressions. To this aim \( \text{last} \) saves the argument of each call into a global repository; if it is called on \( \tau \) and it had already been called on \( \tau \) before, it returns \( \{ \} \):

- \( \text{last}(\epsilon) = \{ \} \)
- \( \text{last}(\tau) = \{ \} \) if \( \text{last} \) had already been called on \( \tau \);
- \( \text{last}(\emptyset ; \tau) = \text{last}(\tau) \cup \{ \emptyset \} \) if \( e(\tau) \);
- \( \text{last}(\tau) = \text{last}(\tau) \) otherwise

- \( \text{last}(\tau_1 \cdot \tau_2) = \text{last}(\tau_2) \)
- \( \text{last}(\tau_1 \lor \tau_2) = \text{last}(\tau_1 \lor \tau_2) = \text{last}(\tau_1) \cup \text{last}(\tau_2) \)

To describe \( \text{DecAMon} \) we first describe the \( \text{DecOne} \) logical predicate, \( \text{DecOne} \subseteq \mathcal{T} \times \mathcal{P}(\mathcal{P}(\text{Ags})) \times \mathcal{P}(\mathcal{P}(\text{Ags})) \).

The way \( \text{DecOne} \) works ensures that the groups of agents in \( \text{Arg} \in \mathcal{P}(\mathcal{P}(\text{Ags})) \) are disjoint, for each \( \text{Arg} \) that can appear as its second or third argument. Given a trace expression \( \tau \in \mathcal{T} \), \( \text{DecOne}(\tau, \text{OldC}, \text{NewC}) \) holds if there exists a constraint store \( C \) s.t. \( C \) satisfies \( \tau \) and \( \text{NewC} = \text{merge}(\text{OldC}, C) \). \( \text{DecOne}(\tau, \text{OldC}, \text{NewC}) \) nondeterministically selects one of the constraint stores that satisfy \( \tau \), let us name it \( C \), and merges it with \( \text{OldC} \) resulting into \( \text{NewC} \). Since \( \text{DecOne} \) must avoid entering loops, it operates like \text{last} keeping track of the already met trace expressions.

- \( \text{DecOne}(\epsilon, \text{OldC}, \text{OldC}) \)
- \( \text{DecOne}(\tau, \text{OldC}, \text{OldC}) \) if \( \text{DecOne} \) had already been called on \( \tau \);
- \( \text{DecOne}(\emptyset ; \tau, \text{OldC}, \text{NewC}) \) if
  i. \( \text{DecOne}(\emptyset ; \tau, \text{OldC}, \text{NewC}) \)
  ii. \( C' \) s.t. \( \text{DecOne}(\tau, \text{OldC}, C') \), \( C \) that satisfies \( \emptyset ; \tau \) and \( \text{NewC} = \text{merge}(C', C) \);
  iii. \( \text{DecOne}(\tau_1 \lor \tau_2, \text{OldC}, \text{NewC}) \)
  (resp. \( \text{DecOne}(\tau_1 \lor \tau_2, \text{OldC}, \text{NewC}) \)) if
  i. \( \text{C_1} \) s.t. \( \text{DecOne}(\tau_1, \text{OldC}, C_1) \), \( \text{C_2} \) s.t. \( \text{DecOne}(\tau_2, C_1, C_2) \), \( C \) that satisfies \( \tau_1 \lor \tau_2 \) (resp. \( \tau_1 \lor \tau_2 \)) and \( \text{NewC} = \text{merge}(C_2; C_1) \);
  ii. \( \text{DecOne}(\tau_1 \lor \tau_2, \text{OldC}, \text{NewC}) \)
  (resp. \( \text{DecOne}(\tau_1 \lor \tau_2, \text{OldC}, \text{NewC}) \)) if
  i. \( \text{C_1} \) s.t. \( \text{DecOne}(\tau_1, \text{OldC}, C_1) \), \( \text{C_2} \) s.t. \( \text{DecOne}(\tau_2, C_1, \text{NewC}) \).

Since \( \land \) and \( | \) do not generate critical points, \( \text{DecOne} \) is just called onto the first branch and the resulting constraint store is passed to the call on the second branch (rule ii); the definition on trace expressions whose outmost operator is either \( \land \) or \( \lor \) is more complex as a further merge with the constraint store generated by these operators is required (rule ii). We recall that \( C_0 \) is \( \tau \) initial constraint store; it contains one set \( \{ \text{ag} \} \) for each agent \( \text{ag} \) involved in \( \tau \).

DEF. 4.1 (MONITORING SAFETY (MS)). A partition of \( \text{Ags} \) \( P \) is Monitoring Safe either if \( \text{DecOne}(\tau, C_{\tau_0}, P) \) holds, or if \( \text{DecOne}(\tau, C_{\tau_0}, P') \) holds and \( P \) can be obtained from \( P' \) by aggregating some groups in it.

We are just one step away from giving the \( \text{DecAMon} \) definition: we need to introduce the \text{findall}(\text{Var}, \text{Goal}, \text{Res}) extra-logical predicate which creates a list \( \text{Res} \) of \( \text{Var} \) instances obtained by backtracking over \( \text{Goal} \). We are ready:
DecAMon(\(\tau\)) = MSs
iff findall(P, DecOne(\(\tau, C_{\tau_0}, P\)), MSs)

We say that a monitor M “checks” a trace expression \(\tau\) if M is in charge for verifying that the events it observes do not violate the current state of the protocol, and the initial state of the protocol is represented by \(\tau\). Theorem 4.1 demonstrates that a partition \(P\) computed by DecOne is monitoring safe.

**Theorem 4.1.** Let \(\tau\) be a trace expression involving agents \(\text{Ag}_s\), let \(C_{\tau_0}\) be \(\tau\) initial constraint store, let \(P = \{Gr_1, Gr_2, \ldots, Gr_N\}\) be one partition computed by DecOne(\(\tau, C_{\tau_0}, P\)), and let \((ev_1, ev_2)\) be a critical point generated by a sub-expression \(\tau_{\text{sub}}\) of \(\tau\).

The centralized monitor \(M_{\text{Ag}_s}\) that checks \(\tau\) detects a violation of \((ev_1, ev_2)\) \(\iff\) there exists \(M_{Gr_i}\) that checks \(\Pi(\tau, Gr_i)\) which detects a violation of \((ev_1, ev_2)\).

**Proof.** See http://decamon.altervista.org/

Theorem 4.1 answers the main research question addressed by this paper and introduced in Section 1: “how to ensure that the system made up of the decentralized monitors detects all and only the same protocol violations that a single centralized monitor observing the MAS would detect.”

### 4.2 Implementation and Experiments

**DecAMon** has been implemented in SWI-Prolog, http://www.swi-prolog.org/. The code can be downloaded from the supplemental material web site and amounts to almost 600 lines. The choice of Prolog was due to many reasons: one-to-one correspondence between the transition and empty rules definitions and their rule-based implementation; built-in support to cyclic terms and to the recognition that a cyclic term has already been met; built-in support to backtracking over goals; availability of Prolog-based tools for trace expressions management.

When the protocol has as many critical points as AIP5, computing all the MSs may require too much time. DecOne can be used instead of DecAMon to compute one MS at a time. As an example, calling DecOne of AIP5 produced the first result in 9 ms, the second one in 8 ms, the third in 1 ms. Although DecOne might not return the best MS according to the designer or the runtime environment needs, its result is guaranteed to be monitoring safe.

If time is not an issue, however, all the MSs can be generated for further post-processing. We implemented the following functionalities which operate on a set of monitoring safe partitions:

1. removing non minimal partitions from the set;
2. selecting those partitions that contain \(N\) agents groups or less (resp. more), where \(N\) is given;
3. selecting those partitions that contain \(M\) singleton agents groups or less (resp. more), where \(M\) is given;
4. selecting those partitions where the agents in the set \(D_i\), given as input in form of a Prolog list, are all disjoint;
5. selecting those partitions where the agents in the set \(T_i\), given as input in form of a Prolog list, are all together.

We run experiments with the protocols introduced in the previous sections, AIP1 to AIP7, plus the following four. We used a MacBook Pro (Retina, 13-inch, Early 2015) with Processor 2.7 GHz Intel Core i5, Memory 8 GB 1867 MHz DDR3, SWI-Prolog version 7.2.3.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>AIP1</th>
<th>AIP4</th>
<th>AIP7</th>
<th>AIP1 norm</th>
<th>AIP7 crit</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MSs (ms)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Table 1:** Experimental results.

AIP₈ = (alice \(\text{submit aamas : aamas \(\rightarrow\) ack alice :} e\)) | (bob \(\text{submit aamas : aamas \(\rightarrow\) ack bob :} e\)) | (carol \(\text{submit aamas : aamas \(\rightarrow\) carol :} e\)) respects the connectedness for sequence condition: the agents can be monitored independently.

AIP₉ = (alice \(\text{submit aamas : chair \(\rightarrow\) review bob :} \) (aamas \(\text{accept alice :} e\)) \(\text{\(\lor\) (aamas \(\text{reject alice :} e\))} \)) | (bob \(\text{submit aamas : chair \(\rightarrow\) alice :} \) (aamas \(\text{accept bob :} e\)) \(\text{\(\lor\) (aamas \(\text{reject bob :} e\))}) demonstrates that DecAMon may return non minimal monitoring safe partitions, which are detected during the post-processing stage. The two only MMSs are \{\text{chair, aamas}, \{alice\}, \{bob\}\} and \{\{alice, bob\}, \{chair\}, \{aamas\}\} but DecAMon also returns \{\{aamas, alice, bob\}, \{chair\}\}, besides others, where AAMAS is uselessly grouped with Alice and Bob.

The other two protocols are variants of the Alternating Bit Protocol (ABP) described in [19]. The ABP is an infinite iteration, where the following constraints have to be satisfied for all occurrences of the interactions:

- the \(n\)-th occurrence of message \(m_1\) must precede the \(n\)-th occurrence of \(m_2\) which in turn must precede the \((n + 1)\)-th occurrence of \(m_3\).
- for \(k \in \{1, 2, 3\}\), the \(n\)-th occurrence of \(m_k\) must precede the \(n\)-th occurrence of the acknowledge \(a_k\), which, in turn, must precede the \((n + 1)\)-th occurrence of \(m_k\).

Because of space constraints, we do not show here the trace expressions corresponding to ABP₉ and ABP₇. The difference between them is that in ABP₉, \(m_1 = bob \rightarrow m_2 alice, m_2 = bob \rightarrow m_3 carol, m_3 = bob \rightarrow m_3 dave, \) and the acknowledges flow in the opposite direction: \(M_{(bob)}\) can monitor all the protocol, as Bob is involved in all the interactions.

In ABP₇, instead, \(m_1 = alice \rightarrow m_1 bob, m_2 = carol \rightarrow m_2 dave, m_3 = emma \rightarrow m_3 frank\) (with their respective acknowledges), so the connectedness for sequence of \(m_1, m_2, \) and \(m_3\) cannot be guaranteed by one monitor alone.
For each protocol we measured the time required by DecAMon to compute its output, the number of MSs computed by DecAMon, the time required to remove the non minimal partitions from DecAMon output, and the number of MMSs. W.r.t. T1.a, we highlight the following aspects:

- the number of computed MSs depends on the trace expression structure and not on its length: AIP₅ and AIP₇ only differ for one operator, but they give different results;
- the number of computed MSs of AIP₄, AIP₈, ABP₉ norm is 1: this means that the monitoring can be fully decentralized, as DecAMon returns only the partition with one singleton group for each agent;
- the number of computed MSs of AIP₉ is different from the number of MMSs: DecAMon may return non minimal partitions.

Since the more groups in the MMS, the better from the decentralization point of view, selecting a MMS with a high number of groups is a good choice for decentralizing as much as possible. Another criterion for preferring a MMS w.r.t. another could be the number of singleton groups, which correspond to agents that can be monitored on their own. Table T1.b shows the results of other post-processing functions, namely the number of MMSs that contain at least 1, 5, 7, 9 agents groups for each protocol, and the number of MMSs that contain at least 1, 5, 7, 9 singleton groups. Although Table T1.b only reports numbers, the post-processing tools return all the partitions that meet the given conditions. The MAS designer or a software agent in charge for the dynamic reconfiguration of the MAS monitoring activity can select one among them and can impose further conditions such as having some agents disjoint or together. By running this tool we discovered for example that there is no MMS of AIP₇ where b, c, d are together, and there are 4224 MMSs where b and m are disjoint.

5. RELATED AND FUTURE WORK

The literature on Distributed Runtime Verification (DRV) is still very limited: the First Workshop on DRV was held at Bertinoro in May 2016, http://www.labri.fr perso/travers/DRV2016/, and the first survey has been published in October 2016 [12]. It references 18 papers only and many of them, such as [24, 25, 27], deal with issues which fall outside the scope of our investigation.

A large amount of contributions are loosely connected with DRV, and investigate the issue of projecting global protocols onto protocol-compliant “skeltons” or “endpoints” in many different areas ranging from MASs [3, 20] to session types [16, 17, 18], from cryptographic protocols [31] to behavioral types for programming languages [2]. Most of the research is focused on the study of well-formedness conditions ensuring that the projection of global protocols can produce correct “enactments”; protocols not meeting such requirements are discarded. The work presented in this paper is entirely devoted to the problem of partial distribution of the protocol verification mechanism even when some well known enactability conditions do not hold.

Singh’s Blindingly Simple Protocol Language (BSPL [34, 35, 36]) is a promising approach for declaratively expressing multiagent protocols. It supports a rich variety of practical protocols and can be realized in a distributed asynchronous architecture where the participating agents act based on local knowledge alone; in this way DRV of declarative protocols is naturally supported. The major difference of our work in comparison to BSPL, is that we face the challenge of DRV of those protocols that do not satisfy the unique point of choice and connectedness for sequence conditions.

Testerink et al. [37, 38] present a formal model for decentralized monitors that supports their formal analysis to face the robustness and security, and a theoretical analysis of distributed runtime norm enforcement. They synthesize the properties that each local monitor is able to verify, expressed in LTL, in order to build a consistent representation of the global state of the world. We do the opposite: we start from a global protocol modeling how the world should behave, and create sub-protocols that involve disjoint groups of agents, in such a way that violations to the global protocol can be discovered by at least one of the monitors in charge for these groups.

The work [33] which is closer to ours addresses the following problem: “given a distributed program D and an LTL₃ property φ, construct a set of monitor processes whose composition with D can evaluate φ at runtime in a sound, complete, and decentralized fashion.” The main differences with our proposal consist in the observed events, which are related to the execution of a program and not to communicative behavior in a MAS, and, most importantly, the use of LTL₃ for specifying system properties; in previous work [8], we have shown that trace expressions are strictly more expressive than LTL₃ when used for runtime verification. Falcone et al. [22] propose an efficient and generalized decentralized monitoring algorithm to detect violation of any regular specification by local monitors without central observation point; also in this case the main difference with our work is the expressive power of the employed formalism for specifications.

Decentralizing the runtime monitoring using DecAMon can prove useful in many situations. The applications we are actually looking at fall in the e-health and well-being domains that we started exploring in the last year [1, 23]. If we have a global protocol describing the expected behavior of a system of communicating low-power wearable devices able to measure vital parameters to check the health conditions of a person, we would like to add lightweight monitors on top of them to monitor only those events “local” to the devices, still being sure that global protocol violations will be detected. In these scenarios, proximity of the monitor to the device is of paramount importance.

For what concerns the time complexity of computing a Minimal Monitoring Safe partition, we suspect that the problem can be reduced to computing a solution to a Minimal Constraint Network [32], recently proven to be NP-hard [26]. If we have a global protocol describing the expected behavior of a system of communicating low-power wearable devices able to measure vital parameters to check the health conditions of a person, we would like to add lightweight monitors on top of them to monitor only those events “local” to the devices, still being sure that global protocol violations will be detected. In these scenarios, proximity of the monitor to the device is of paramount importance.

For what concerns the time complexity of computing a Minimal Monitoring Safe partition, we suspect that the problem can be reduced to computing a solution to a Minimal Constraint Network [32], recently proven to be NP-hard [26]. If we have a global protocol describing the expected behavior of a system of communicating low-power wearable devices able to measure vital parameters to check the health conditions of a person, we would like to add lightweight monitors on top of them to monitor only those events “local” to the devices, still being sure that global protocol violations will be detected. In these scenarios, proximity of the monitor to the device is of paramount importance.
REFERENCES


