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 "connectedness 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 *Dec*entralizing the *A*gent system *Mon*itoring 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".

*Corresponding author, PhD student at DIBRIS.

Appears in: *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

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 intuition behind *DecAMon* by means of examples; Section 4 provides the technical details of *DecAMon* design and implementation and presents the experimental results; Section 5 analyzes the related work and concludes. The proof of Theorem 4.1, the *DecAMon* code, and the results of those experiments that, for space constraints, have not been included in this paper, can be found at `http://decamon.altervista.org/`.

2. MOTIVATION

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 by the evening, Alice will contact their supervisor to meet, otherwise Carol will ask Bob to meet, to fix the problems.

If Alice and Carol are modeled as software agents, the resulting global agent interaction protocol, AIP_1, can be represented by $alice \stackrel{meet}{\Longrightarrow} dave : \epsilon \ \vee \ carol \stackrel{meet}{\Longrightarrow} bob : \epsilon$ where $ag1 \stackrel{msg}{\Longrightarrow} ag2$ stands for "there is an interaction between $ag1$ and $ag2$, with exchanged message msg". The : operator stands for sequential composition of an interaction and the remainder of the protocol, ϵ stands for the empty protocol, and $\vee$ stands for nondeterministic choice where, if one branch is selected, the other branch cannot be selected any longer (mutual exclusivity).

Once the paper is ready, Alice and Carol decide that Alice will make a first submission in the morning and then she will make a last check and, eventually, modify and re-submit the paper. Carol will send the final version to their supervisor in the evening. The resulting protocol can be modeled by

$$\mathrm{AIP}_2 = alice \stackrel{submit}{\Longrightarrow} aamas : (alice \stackrel{submit}{\Longrightarrow} aamas : \epsilon \ \vee \ \epsilon) \ \cdot \ carol \stackrel{submitted}{\Longrightarrow} dave : \epsilon$$

where Alice makes one or two submissions and after ($\cdot$ stands for protocol concatenation) Carol sends *submitted* to Dave.

Even if AIP_1 and AIP_2 are simple and realistic, they have two major problems that are well known in the behavioral types community [29, 30] and which prevent them from being used for fully decentralized monitoring.

The first problem is that these protocols are too abstract. Although their meaning can be easily grasped, what does actually mean for two agents to interact? Let us consider this simple example: $alice \stackrel{m1}{\Longrightarrow} bob : alice \stackrel{m2}{\Longrightarrow} carol : \epsilon$. Once Alice has sent $m1$ to Bob, could she immediately send $m2$ to Carol, without worrying if Bob received $m1$, or not? This problem is related with the granularity of interactions, which describe message sending and receiving as an atomic action. Since, in many cases, communication is asynchronous, assuming interaction atomicity is not always the case. We should decouple the sending event and the reception event, in order to provide a more precise description of the system. By using $ag1_{\langle \stackrel{msg}{\Longrightarrow} ag2\rangle}$ to denote "$ag1$ sends msg to $ag2$" and ${}_{\langle ag1 \stackrel{msg}{\Longrightarrow}\rangle} ag2$ to denote "$ag2$ receives msg from $ag1$", we can describe 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*), $\mathrm{CAIP}_1 = a_{\langle \stackrel{m1}{\Longrightarrow} b\rangle} : {}_{\langle a \stackrel{m1}{\Longrightarrow}\rangle} b : a_{\langle \stackrel{m2}{\Longrightarrow} c\rangle} : {}_{\langle a \stackrel{m2}{\Longrightarrow}\rangle} c : \epsilon$, and an asynchronous one, where sending comes first, the corresponding reception comes after, but other events could take place in between, $\mathrm{CAIP}_2 = a_{\langle \stackrel{m1}{\Longrightarrow} b\rangle} : ({}_{\langle a \stackrel{m1}{\Longrightarrow}\rangle} b : \epsilon \ | \ a_{\langle \stackrel{m2}{\Longrightarrow} c\rangle} : {}_{\langle a \stackrel{m2}{\Longrightarrow}\rangle} c : \epsilon)$[1].
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 τ_1 and τ_2 are independent.

The second problem is related with monitoring the behavior of each agent ag w.r.t. the *view that ag has of the global protocol*, namely, the global protocol restricted to those events that involve ag. Moving from a global protocol 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 $\Pi : \mathcal{T} \times \mathcal{P}(Ags) \rightarrow \mathcal{T}$, 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_{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_{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_{alice}, M_{carol}, M_{bob}, M_{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_{carol} sees is

$$carol \stackrel{submitted}{\Longrightarrow} dave : \epsilon$$

[1]We name these protocols $C\mathrm{AIP}_1$ and $C\mathrm{AIP}_2$ to stress that they are "*C*oncrete".

On the other hand, the portion seen by M_{alice} is

$$alice \overset{submit}{\Longrightarrow} aamas : (alice \overset{submit}{\Longrightarrow} aamas : \epsilon \ \vee \ \epsilon)$$

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 terminology, 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 $AIP_4 = alice \overset{submit}{\Longrightarrow} aamas : (alice \overset{sumbitted}{\Longrightarrow} dave : \epsilon \ | \ aamas \overset{ack}{\Longrightarrow} alice : \epsilon)$ could be violated by Alice submitting the paper twice. However, the second $alice \overset{submit}{\Longrightarrow} aamas$ interaction would violate both $\Pi(AIP_4, \{alice\})$ and $\Pi(AIP_4, \{aamas\})$ (the projections of the global protocol AIP_4 onto $\{alice\}$ and $\{aamas\}$, respectively), so at least one decentralized monitor between M_{alice} and M_{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 Ags of agents in the MAS, *DecAMon* faces the partial decentralization problem by computing a set of "Monitoring Safe (MS) partitions" of 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 *alice*, *bob*, *carol*, and *dave*. If $\{\{alice, carol\}, \{bob\}, \{dave\}\}$ is a MS partition, then *alice* and *carol* must be monitored by the same monitor $M_{\{alice,carol\}}$, whereas *bob* and *dave* may be monitored by distinct monitors. This does not mean that having one monitor $M_{\{alice,carol\}}$ for *alice* and *carol* and one $M_{\{bob,dave\}}$ for *bob* and *dave* (to be monitored together), or one single monitor $M_{\{alice,bob,carol,dave\}}$ 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 *alice* and *carol*, it will be able to detect violations of mutual exclusivity between $alice \overset{meet}{\Longrightarrow} dave$ and $carol \overset{meet}{\Longrightarrow} bob$.

In a similar way, one MS partition of the agents involved in AIP_2 is $\{\{alice, dave\}, \{aamas\}, \{carol\}\}$: if the same monitor is in charge for both *alice* and *dave*, it can verify that the interaction involving *dave* (and *carol*) takes place after the interactions involving *alice* (and *aamas*).

Intuition 3.1 (Monitoring Safety (MS)). *A partition of 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, $\{Ags\}$. On the other hand, if each agent $ag_i \in Ags$, with $i \in \{1, ..., n\}$ can be monitored independently from the others, *DecAMon* will output $\{\{ag_1\}, \{ag_2\}, \{ag_3\}, ..., \{ag_n\}\}$.

DecAMon agnosticism 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

$$involved(ag1 \overset{msg}{\Longrightarrow} ag2) = \{ag1, ag2\}$$
$$involved(ag1_{\langle \overset{msg}{\Longrightarrow} ag2 \rangle}) = \{ag1\}$$
$$involved({}_{\langle ag1 \overset{msg}{\Longrightarrow} \rangle} ag2) = \{ag2\}.$$

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 ${}_{\langle ag1 \overset{msg}{\Longrightarrow} \rangle} ag2$ and viceversa.

Continuing the AIP_1 example, the other MSs are

$$\{\{alice\}, \{carol\}, \{bob, dave\}\}$$
$$\{\{alice, bob\}, \{carol\}, \{dave\}\}$$
$$\{\{alice\}, \{carol, dave\}, \{bob\}\}$$

If ev_1 and ev_2 are joined by an $\vee$ operator in the AIP like $alice \overset{meet}{\Longrightarrow} dave$ and $carol \overset{meet}{\Longrightarrow} bob$ in AIP_1, and $involved(ev_1)$ has empty intersection with $involved(ev_2)$, one agent $ag1 \in involved(ev_1)$ must be monitored together with one agent $ag2 \in involved(ev_2)$ to ensure that mutual exclusivity becomes verifiable. In a similar way, if ev_1 and ev_2 are two sequential events like $alice \overset{submit}{\Longrightarrow} aamas$ and $carol \overset{submitted}{\Longrightarrow} dave$ in AIP_2, and $involved(ev_1)$ has empty intersection with $involved(ev_2)$, one agent $ag1 \in involved(ev_1)$ must be monitored together with one agent $ag2 \in involved(ev_2)$ in order to verify the correct sequentiality of ev_1 and ev_2. In both cases, if there exists one agent $agI \in involved(ev_1) \cap involved(ev_2)$ 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 $\mathcal{E}$ a fixed universe of events ($carol \overset{submit}{\Longrightarrow} dave$ is an event) and by $\mathcal{ET}$ a fixed universe of event types each denoting a subset of events in $\mathcal{E}$ ($SubmitType = \{alice \overset{submit}{\Longrightarrow} aamas, carol \overset{submit}{\Longrightarrow} dave\}$ 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[2] 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 $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. 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.

[2]W.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.

• 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 ev matches the event type ϑ ($ev \in \vartheta$), 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.

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 $\tau = alice \stackrel{submit}{\Longrightarrow} aamas : (\tau \vee \epsilon) \cdot carol \stackrel{submitted}{\Longrightarrow} dave : \epsilon$ 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 $alice \stackrel{submit}{\Longrightarrow} aamas$ and c the event $carol \stackrel{submitted}{\Longrightarrow} dave$, the denotational semantics of τ is the set of event traces $\{ac, aac, aaac, \ldots, a^n c, a^\omega\}$ where a^n stands for a trace containing n instances of the event $alice \stackrel{submit}{\Longrightarrow} aamas$ and a^ω stands for an infinite trace of $alice \stackrel{submit}{\Longrightarrow} aamas$ 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 $first(\tau)$ returns all the first events of τ and $last(\tau)$ all its last events. For example, $first(a : \epsilon \vee b : c : \epsilon) = \{a, b\}$ and $last(a : \epsilon \vee b : c : \epsilon) = \{a, c\}$. 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 τ iff τ_{sub} is a sub-expression of τ such that*
• $\tau_{sub} = \tau_1{\vee}\tau_2$ *and* $\exists ev_1 \in first(\tau_1)$, $\exists ev_2 \in first(\tau_2)$ *s.t.* $involved(ev_1) \cap involved(ev_2) = \emptyset$, *or*
• $\tau_{sub} = ev_1{:}\tau_2$ *and* $\exists ev_2 \in first(\tau_2)$ *s.t.* $involved(ev_1) \cap involved(ev_2) = \emptyset$, *or*
• $\tau_{sub} = \tau_1{\cdot}\tau_2$ *and* $\exists ev_1 \in last(\tau_1)$, $\exists ev_2 \in first(\tau_2)$ *s.t.* $involved(ev_1) \cap involved(ev_2) = \emptyset$.

We say that τ_{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 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 satisfies monitoring safety any longer.*

For generating a set of MSs, *DecAMon* exploits *merge*:
$$merge : \mathcal{P}(\mathcal{P}(Ags)) \times \mathcal{P}(\mathcal{P}(Ags)) \rightarrow \mathcal{P}(\mathcal{P}(Ags))$$

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 *OldC* models the existing agent grouping constraints: we name them "constraint stores". The result of *merge* is a new constraint store *NewC* where both the constraints in *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 $NewC \in \mathcal{P}(\mathcal{P}(Ags))$ will be disjoint if the groups of agents in C were disjoints, and the groups of agents in *OldC* were. Let us introduce *merge* by means of an example: the idea behind $merge(\{\{ag1, ag2\}\}, \{\{ag1, ag3\}, \{ag4, ag5\}\})$ is to add the new constraint "agents $ag1$ and $ag2$ must be monitored together" to the constraint store $\{\{ag1, ag3\}, \{ag4, ag5\}\}$ stating that $ag1$ and $ag3$ must be monitored together, as well as $ag4$ and $ag5$. The only constraint store resulting from this merge is $NewC = \{\{ag1, ag2, ag3\}, \{ag4, ag5\}\}$, where both the previous and the new constraints are respected. The amount of agents that are constrained to be monitored together is minimized: in $NewC$, $ag1$ and $ag4$ can be monitored independently as there is no reason to group them. The constraint store $NewC' = \{\{ag1, ag2, ag3, ag4, ag5\}\}$ 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 τ_1 and each of those resulting from τ_2. Since the constraint stores deriving from τ_1 and τ_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 MMSs as a refinement of the *DecAMon* output, by removing those MSs that add useless constraints to other MSs. The global minimality property can be obtained either via this post-processing activity where each returned MS is compared with all the others, or by making *merge* more complex (*merge* would need to know all the possible MSs for each trace expression branch to discard the overlapping ones and return only minimal MSs). We opted for the first solution.

Let us consider another example: if we had to compute $merge(\{\{ag1, ag2, ag4\}\}, \{\{ag1, ag3\}, \{ag2, ag5\}\})$ the only possible result would be to merge $\{ag1, ag3\}$ and $\{ag2, ag5\}$ 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 $\{\{ag1, ag2, ag3, ag4, ag5\}\}$.

Let us consider the more complex protocol AIP_5 defined as $(ab{:}bc{:}\epsilon \mid de{:}ef{:}\epsilon \mid gh{:}hi{:}\epsilon) \cdot (jk{:}\epsilon \mid lm{:}\epsilon \mid no{:}\epsilon)$ where ab stands for $a \xRightarrow{ab} b$, bc stands for $b \xRightarrow{bc} c$, and so on.

DecAMon starts exploring AIP_5 looking for critical points. The outmost AIP_5 operator is a concatenation, which may generate critical points. *DecAMon* computes all the last events in $\tau_1 = (ab{:}bc{:}\epsilon \mid de{:}ef{:}\epsilon \mid gh{:}hi{:}\epsilon)$ and all the first events in $\tau_2 = (jk{:}\epsilon \mid lm{:}\epsilon \mid no{:}\epsilon)$. They turn out to be $last(\tau_1) = \{bc, ef, hi\}$ and $first(\tau_2) = \{jk, lm, no\}$.

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, jk), (bc, lm), (bc, no), (ef, jk), (ef, lm), (ef, no), (hi, jk), (hi, lm), (hi, no), are all the critical points generated by the outmost $\cdot$ in AIP_5.

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, $C5_1 = \{\{b, e, h, j, l, n\}\}$ satisfies $\text{AIP}_5 = \tau_1 \cdot \tau_2$ 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 $C5_2 = \{\{b, k, m, o\}, \{e, h, j, l, n\}\}$, satisfies AIP_5: b is grouped with k, m, and o hence satisfying (bc, jk), (bc, 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 $C5_3 = \{\{c, k\}, \{b, m, o\}, \{e, h, j, l, n\}\}$: $\{c, k\}$ satisfies (bc, jk); $\{b, m, o\}$ satisfies (bc, lm) 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_5: for example, no group satisfies (bc, lm).

We define $C\tau_0$ as the constraint store that contains all and only one singleton set $\{ag\}$ for each agent ag involved in τ. Given the initial constraint store $C5_0$ for the protocol AIP_5, *DecAMon* merges $C5_0$ with one of the constraint stores $C5_i$ that satisfy AIP_5, selected on a nondeterministic basis. Then, it recursively explores the components τ_1 and τ_2 of AIP_5 and adds the newly discovered constraints to the previously computed constraint store. The sequences $ab{:}bc{:}\epsilon$, $de{:}ef\epsilon$ and $gh{:}hi{:}\epsilon$ in τ_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 τ_1. In a similar way, no new constraints are generated because of τ_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: $C5_1$, $C5_2$, $C5_3$, $C5_4$, together with other 5628 possible initial constraint stores, are the MSs output by *DecAMon*!

If τ_1 were $(ab{:}cd{:}\epsilon \mid ef{:}gh{:}\epsilon)$ (AIP_6) the new constraints $\{\{a, c\}, \{e, g\}\}$ (or $\{\{a, d\}, \{f, g\}\}$, ...) due to connectedness for sequence violation should have been merged with $C5_i$ giving a different (and smaller) final set of MSs.

If τ_1 were $(ab{:}cd{:}\epsilon \vee ef{:}gh{:}\epsilon)$ (AIP_7) 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 $\text{CAIP}_1 = a_{\langle \xRightarrow{m1} b\rangle} : {}_{\langle a \xRightarrow{m1}\rangle} b : a_{\langle \xRightarrow{m2} c\rangle} : {}_{\langle a \xRightarrow{m2}\rangle} c : \epsilon$. As anticipated, *DecAMon* works exactly in the same way, provided it can compute the *involved* function. CAIP_1 can be seen as $a_{\langle \xRightarrow{m1} b\rangle} : \tau_1$. Its outmost operator is the first prefix with $\{a_{\langle \xRightarrow{m1} b\rangle}\}$ at its left, $first(\tau_1) = \{{}_{\langle a \xRightarrow{m1}\rangle} b\}$ and these two events share no involved agents: $(a_{\langle \xRightarrow{m1} b\rangle}, {}_{\langle a \xRightarrow{m1}\rangle} b)$ 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 $\{\{a, b\}, \{c\}\}$. Now *DecAMon* is called on $\tau_1 = {}_{\langle a \xRightarrow{m1}\rangle} b : a_{\langle \xRightarrow{m2} c\rangle} : {}_{\langle a \xRightarrow{m2}\rangle} c : \epsilon$ generating the new constraint store $\{\{a, c\}\}$ which must be merged with $\{\{a, b\}, \{c\}\}$ leading to $\{\{a, b, c\}\}$. In the end, all the three agents must be monitored together. This is correct: how can M_b alone verify that when b receives $m1$ 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 $\delta \subseteq \mathcal{T} \times \mathcal{E} \times \mathcal{T}$. $\tau_1 \xrightarrow{ev} \tau_2$ means $(\tau_1, ev, \tau_2) \in \delta$. If the trace expression τ_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 $\tau_1 \xrightarrow{ev} \tau_2$; in such a case, τ_2 specifies the next valid state of the protocol after event ev takes place. Otherwise, the event ev is not valid in τ_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].

Figure 1 defines the inductive rules for δ. While δ defines

$$
(\text{prefix})\,\frac{}{\vartheta{:}\tau \xrightarrow{ev} \tau}\; ev\in\vartheta
\qquad
(\text{or-l})\,\frac{\tau_1 \xrightarrow{ev} \tau_1'}{\tau_1\vee\tau_2 \xrightarrow{ev} \tau_1'}
\qquad
(\text{or-r})\,\frac{\tau_2 \xrightarrow{ev} \tau_2'}{\tau_1\vee\tau_2 \xrightarrow{ev} \tau_2'}
\qquad
(\text{and})\,\frac{\tau_1 \xrightarrow{ev} \tau_1' \quad \tau_2 \xrightarrow{ev} \tau_2'}{\tau_1\wedge\tau_2 \xrightarrow{ev} \tau_1'\wedge\tau_2'}
\qquad
(\text{shuffle-l})\,\frac{\tau_1 \xrightarrow{ev} \tau_1'}{\tau_1|\tau_2 \xrightarrow{ev} \tau_1'|\tau_2}
$$

$$
(\text{shuffle-r})\,\frac{\tau_2 \xrightarrow{ev} \tau_2'}{\tau_1|\tau_2 \xrightarrow{ev} \tau_1|\tau_2'}
\qquad
(\text{cat-l})\,\frac{\tau_1 \xrightarrow{ev} \tau_1'}{\tau_1\cdot\tau_2 \xrightarrow{ev} \tau_1'\cdot\tau_2}
\qquad
(\text{cat-r})\,\frac{\tau_2 \xrightarrow{ev} \tau_2'}{\tau_1\cdot\tau_2 \xrightarrow{ev} \tau_2'}\;\epsilon(\tau_1)
\qquad
(\text{cond-t})\,\frac{\tau \xrightarrow{ev} \tau'}{\vartheta\gg\tau \xrightarrow{ev} \vartheta\gg\tau'}\; ev\in\vartheta
\qquad
(\text{cond-f})\,\frac{}{\vartheta\gg\tau \xrightarrow{ev} \vartheta\gg\tau}\; ev\not\in\vartheta
$$

Figure 1: Operational semantics of trace expressions

$$
(\epsilon\text{-empty})\,\frac{}{\epsilon(\epsilon)}
\qquad
(\epsilon\text{-or-l})\,\frac{\epsilon(\tau_1)}{\epsilon(\tau_1\vee\tau_2)}
\qquad
(\epsilon\text{-or-r})\,\frac{\epsilon(\tau_2)}{\epsilon(\tau_1\vee\tau_2)}
\qquad
(\epsilon\text{-shuffle})\,\frac{\epsilon(\tau_1)\quad\epsilon(\tau_2)}{\epsilon(\tau_1|\tau_2)}
\qquad
(\epsilon\text{-cat})\,\frac{\epsilon(\tau_1)\quad\epsilon(\tau_2)}{\epsilon(\tau_1\cdot\tau_2)}
\qquad
(\epsilon\text{-and})\,\frac{\epsilon(\tau_1)\quad\epsilon(\tau_2)}{\epsilon(\tau_1\wedge\tau_2)}
\qquad
(\epsilon\text{-cond})\,\frac{\epsilon(\tau)}{\epsilon(\vartheta\gg\tau)}
$$

Figure 2: Empty trace containment

the non empty traces of a trace expression, the predicate $\epsilon(\_)$, inductively defined by the rules in Figure 2, defines the trace expressions that contain the empty trace ϵ and hence may terminate. If $\epsilon(\tau)$ holds, then the empty trace is a valid trace for τ. Both figures are taken from [8].

A SWI-Prolog implementation of these rules can be downloaded from `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, `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. to 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 $first$ does not need to take cycles – which are due to trace expressions recursive definitions – into account; in fact, contractiveness ensures that, while exploring a trace expression following its syntactical structure, a prefix operator will be met in a finite number of steps.

– $first(\epsilon) = \{\}$
– $first(\vartheta{:}\tau) = \{\vartheta\}$
– $first(\tau_1{\cdot}\tau_2) = first(\tau_1) \cup first(\tau_2)$ if $\epsilon(\tau_1)$;
  $first(\tau_1{\cdot}\tau_2) = first(\tau_1)$ otherwise
– $first(\tau_1\wedge\tau_2) = first(\tau_1\vee\tau_2) = first(\tau_1|\tau_2) =$
  $first(\tau_1) \cup first(\tau_2)$.

The definition of $last$ is more complex because it must not recall itself in case of cyclic trace expressions and contractiveness is not enough to avoid entering a loop. For example, $\tau = ev{:}\tau$ is contractive, but we must have plenty of time and patience if we are going to look for its last element! In this case, $last$ 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 $last$ saves the argument of each call into a global repository; if it is called on τ and it had already been called on τ before, it returns $\{\}$:

– $last(\epsilon) = \{\}$
– $last(\tau) = \{\}$ if $last$ had already been called on τ;
  otherwise, the following rules apply:
  – $last(\vartheta{:}\tau) = last(\tau) \cup \{\vartheta\}$ if $\epsilon(\tau)$;
    $last(\vartheta{:}\tau) = last(\tau)$ otherwise
  – $last(\tau_1{\cdot}\tau_2) = last(\tau_2)$
  – $last(\tau_1\wedge\tau_2) = last(\tau_1\vee\tau_2) = last(\tau_1|\tau_2) =$
    $last(\tau_1) \cup last(\tau_2)$.

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

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

• $DecOne(\epsilon, OldC, OldC)$

• $DecOne(\tau, OldC, OldC)$ if $DecOne$ had already been called on τ; otherwise, the following rules apply:

i. $DecOne(\vartheta{:}\tau, OldC, NewC)$ iff
   $\exists\, C'$ s.t. $DecOne(\tau, OldC, C')$,
   $\exists\, C$ that satisfies $\vartheta{:}\tau$, and
   $NewC = merge(C', C)$;
ii. $DecOne(\tau_1{\cdot}\tau_2, OldC, NewC)$
   (resp. $DecOne(\tau_1\vee\tau_2, OldC, NewC)$) iff
   $\exists\, C_1$ s.t. $DecOne(\tau_1, OldC, C_1)$,
   $\exists\, C_2$ s.t. $DecOne(\tau_2, C_1, C_2)$,
   $\exists\, C$ that satisfies $\tau_1{\cdot}\tau_2$ (resp. $\tau_1\vee\tau_2$) and
   $NewC = merge(C_2, C)$;
iii. $DecOne(\tau_1\wedge\tau_2, OldC, NewC)$
   (resp. $DecOne(\tau_1|\tau_2, OldC, NewC)$) iff
   $\exists\, C_1$ s.t. $DecOne(\tau_1, OldC, C_1)$,
   $\exists\, C_2$ s.t. $DecOne(\tau_2, C_1, NewC)$.

Since $\wedge$ and $|$ do not generate critical points, $DecOne$ is just called onto the first branch and the resulting constraint store is passed to the call on the second branch (rule iii); the definition on trace expressions whose outmost operator is either $\cdot$ or $\vee$ is more complex as a further merge with the constraint store generated by these operators is required (rule ii). We recall that $C\tau_0$ is τ initial constraint store: it contains one set $\{ag\}$ for each agent ag involved in τ.

Def. 4.1 (Monitoring Safety (MS)). *A partition of Ags P is Monitoring Safe either if $DecOne(\tau,\ C\tau_0,\ P)$ holds, or if $DecOne(\tau,\ C\tau_0,\ P')$ holds and P can be obtained from P' by aggregating some groups in it.*

We are just one step away from giving the $DecAMon$ definition: we need to introduce the $findall(Var,\ Goal,\ Res)$ extra-logical predicate which creates a list Res of Var instances obtained by backtracking over $Goal$. We are ready:

$$DecAMon(\tau) = MSs$$
$$\text{iff } findall(P,\ DecOne(\tau,\ C\tau_0,\ P),\ MSs)$$

We say that a monitor M "checks" a trace expression τ 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 τ. Theorem 4.1 demonstrates that a partition P computed by $DecOne$ is monitoring safe.

THEOREM 4.1. *Let τ be a trace expression involving agents Ags, let $C\tau_0$ be τ initial constraint store, let $P = \{Gr1, Gr2, ..., GrN\}$ 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 τ_{sub} of τ.*

The centralized monitor M_{Ags} that checks τ detects a violation of (ev_1, ev_2) $\iff$ there exists M_{GrI} that checks $\Pi(\tau, GrI)$ 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 AIP_5, 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 AIP_5 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, given as input in form of a Prolog list, are all disjoint;
5. selecting those partitions where the agents in the set T, given as input in form of a Prolog list, are all together.

We run experiments with the protocols introduced in the previous sections, AIP_1 to AIP_7, 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.

Protocol	*Dec* (ms)	*Dec* (♯)	MMS (ms)	MMS (♯)
aip1	2	4	2	4
aip2	1	4	1	4
aip3	1	4	1	4
aip4	1	1	1	1
aip5	122400	5632	74711	5632
aip6	308	256	139	256
aip7	309	128	37	128
aip8	1	1	1	1
aip9	1	16	1	2
abp_norm	14	1	1	1
abp_crit	8	16	1	16

T1.a: *DecAMon* execution time (*Dec* (**ms**));
MSs returned by *DecAMon* (*Dec* (♯));
execution time of the tool for removing non-minimal MSs (**MMS (ms)**);
computed MMSs (**MMS (♯)**).

Protocol	$\geq 1g$	$\geq 5g$	$\geq 7g$	$\geq 9g$	$\geq 1s$	$\geq 5s$	$\geq 7s$	$\geq 9s$
aip1	4	0	0	0	4	0	0	0
aip2	4	0	0	0	4	0	0	0
aip3	4	0	0	0	4	0	0	0
aip4	1	0	0	0	1	0	0	0
aip5	5632	5632	1600	64	5632	1600	64	64
aip6	256	256	128	0	256	128	128	0
aip7	128	128	128	0	128	128	128	0
aip8	1	0	0	0	1	0	0	0
aip9	2	0	0	0	2	0	0	0
abp_norm	1	0	0	0	1	0	0	0
abp_crit	16	0	0	0	16	0	0	0

T1.b: Number of MMSs that contain at least 1, 5, 7, 9 agents groups (columns $\geq 1g$, $\geq 5g$, $\geq 7g$, $\geq 9g$).
Number of MMSs that contain at least 1, 5, 7, 9 singleton groups (columns $\geq 1s$, $\geq 5s$, $\geq 7s$, $\geq 9s$).

Table 1: Experimental results.

$$\begin{aligned}\text{AIP}_8 = {} & (alice \overset{submit}{\Longrightarrow} aamas : aamas \overset{ack}{\Longrightarrow} alice : \epsilon) \ | \\ & (bob \overset{submit}{\Longrightarrow} aamas : aamas \overset{ack}{\Longrightarrow} bob : \epsilon) \ | \\ & (carol \overset{submit}{\Longrightarrow} aamas : aamas \overset{ack}{\Longrightarrow} carol : \epsilon)\end{aligned}$$

respects the connectedness for sequence condition: the agents can be monitored independently.

$$\begin{aligned}\text{AIP}_9 = {} & (alice \overset{submit}{\Longrightarrow} aamas : chair \overset{review}{\Longrightarrow} bob : \\ & \quad (aamas \overset{accept}{\Longrightarrow} alice : \epsilon) \lor (aamas \overset{reject}{\Longrightarrow} alice : \epsilon)) \\ & | \ (bob \overset{submit}{\Longrightarrow} aamas : chair \overset{review}{\Longrightarrow} alice : \\ & \quad (aamas \overset{accept}{\Longrightarrow} bob : \epsilon) \lor (aamas \overset{reject}{\Longrightarrow} bob : \epsilon))\end{aligned}$$

demonstrates that *DecAMon* may return non minimal monitoring safe partitions, which are detected during the post-processing stage. The two only MMSs are $\{\{chair, aamas\}, \{alice\}, \{bob\}\}$ 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 `m1` must precede the n-th occurrence of `m2` which in turn must precede the n-th occurrence of `m3`.

– For $k \in \{1, 2, 3\}$, the n-th occurrence of $\mathtt{m}k$ must precede the n-th occurrence of the acknowledge $\mathtt{a}k$, which, in turn, must precede the $(n+1)$-th occurrence of $\mathtt{m}k$.

Because of space constraints, we do not show here the trace expressions corresponding to ABP_{norm} and ABP_{crit}. The difference between them is that in ABP_{norm}, $\mathtt{m1} = bob \overset{m1}{\Longrightarrow} alice$, $\mathtt{m2} = bob \overset{m2}{\Longrightarrow} carol$, $\mathtt{m3} = bob \overset{m3}{\Longrightarrow} dave$, and the acknowledges flow in the opposite direction: $\text{M}_{\{bob\}}$ can monitor all the protocol, as Bob is involved in all the interactions.

In ABP_{crit}, instead, $\mathtt{m1} = alice \overset{m1}{\Longrightarrow} bob$, $\mathtt{m2} = carol \overset{m2}{\Longrightarrow} dave$, $\mathtt{m3} = emma \overset{m3}{\Longrightarrow} frank$ (with their respective acknowledges), so the connectedness for sequence of `m1`, `m2`, and `m3` 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: AIP_6 and AIP_7 only differ for one operator, but they give different results;

– the number of computed MSs of AIP_4, AIP_8, ABP_{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 AIP_9 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 AIP_5 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 "skeletons" 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 LTL_3 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 LTL_3 for specifying system properties; in previous work [8], we have shown that trace expressions are strictly more expressive than LTL_3 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]. For the applications we have in mind, the need for decentralizing a protocol for monitoring purposes arises only seldom, so the (possibly) high complexity of *DecAMon* can be tolerated. Experiments on several protocols have empirically shown that, once the protocol has been decentralized, the time complexity of monitoring is linear in the trace length, and does not depend on the number of involved agents.

We leave for future developments the investigation of suitable heuristics for boosting the efficiency of *DecAMon* when a MAS partition must be recomputed very often; other interesting research directions include the extension of *DecAMon* to deal with parametric trace expressions [9], and its exploitation in other promising application domains, such as ambient intelligence systems [5] and traffic monitoring.

REFERENCES

[1] F. Aielli, D. Ancona, P. Caianiello, S. Costantini, G. D. Gasperis, A. D. Marco, A. Ferrando, and V. Mascardi. FRIENDLY & KIND with your health: Human-friendly knowledge-intensive dynamic systems for the e-health domain. In J. Bajo, M. J. Escalona, S. Giroux, P. Hoffa-Dabrowska, V. Julián, P. Novais, N. S. Pi, R. Unland, and R. A. Silveira, editors, *Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection - International Workshops of PAAMS 2016. Proceedings*, volume 616 of *Communications in Computer and Information Science*, pages 15–26. Springer, 2016.

[2] 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. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, and N. Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

[3] D. Ancona, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert. Efficient verification of MASs with projections. In F. Dalpiaz, J. Dix, and M. B. van Riemsdijk, editors, *Engineering Multi-Agent Systems: Second International Workshop, EMAS 2014, Revised Selected Papers*, pages 246–270. Springer, 2014.

[4] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Global protocols as first class entities for self-adaptive agents. In G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, editors, *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015. Proceedings*, pages 1019–1029. ACM, 2015.

[5] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Runtime verification of fail-uncontrolled and ambient intelligence systems: A uniform approach. *Intelligenza Artificiale*, 9(2):131–148, 2015.

[6] D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. MAS-DRiVe: a practical approach to decentralized runtime verification of agent interaction protocols. In C. Santoro, F. Messina, and M. D. Benedetti, editors, *From Objects to Agents, 17th Workshop, WOA 2016. Proceedings*, volume 1664 of *CEUR Workshop Proceedings*, pages 35–43. CEUR-WS.org, 2016.

[7] D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In M. Baldoni, L. A. Dennis, V. Mascardi, and W. W. Vasconcelos, editors, *Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012. Revised Selected Papers*, volume 7784 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 2013.

[8] D. Ancona, A. Ferrando, and V. Mascardi. *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, chapter Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification, pages 47–64. Springer, 2016.

[9] D. Ancona, A. Ferrando, and V. Mascardi. Parametric runtime verification of multiagent systems. In S. Das, E. Durfee, K. Larson, and M. Winikoff, editors, *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2017. Proceedings.* IFAAMAS, 2017.

[10] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011.

[11] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

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

[13] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

[14] D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE and Jason multiagent systems with Prolog. In L. Giordano, V. Gliozzi, and G. L. Pozzato, editors, *Italian Conference on Computational Logic, 29th edition, CILC 2014. Proceedings*, volume 1195 of *CEUR Workshop Proceedings*, pages 319–323. CEUR-WS.org, 2014.

[15] D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE multiagent systems. In D. Camacho, L. Braubach, S. Venticinque, and C. Badica, editors, *Intelligent Distributed Computing VIII - 8th International Symposium, IDC 2014. Proceedings*, volume 570 of *Studies in Computational Intelligence*, pages 81–91. Springer, 2014.

[16] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, June 2012.

[17] M. Coppo, M. Dezani-Ciancaglini, L. Padovani, and N. Yoshida. A gentle introduction to multiparty asynchronous session types. In M. Bernardo and E. B. Johnsen, editors, *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Advanced Lectures*, volume 9104 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2015.

[18] R. Corin, P. Deniélou, C. Fournet, K. Bhargavan, and J. J. Leifer. Secure implementations for typed session abstractions. In *20th IEEE Computer Security Foundations Symposium, CSF 2007*, pages 170–186. IEEE Computer Society, 2007.

[19] P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In H. Seidl, editor, *Programming Languages and Systems, 21st European Symposium on Programming, ESOP 2012. Proceedings*, pages 194–213. Springer, 2012.

[20] N. Desai and M. P. Singh. Protocol-based business process modeling and enactment. In *Proceedings of the IEEE International Conference on Web Services*

(ICWS'04), pages 35–42. IEEE Computer Society, 2004.

[21] N. Desai and M. P. Singh. On the enactability of business protocols. In D. Fox and C. P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008*, pages 1126–1131. AAAI Press, 2008.

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

[23] A. Ferrando, D. Ancona, and V. Mascardi. Monitoring patients with hypoglycemia using self-adaptive protocol-driven agents: A case study. In M. Baldoni, J. P. Müller, I. Nunes, and R. Zalila-Wenkstern, editors, *Engineering Multi-Agent Systems - 4th International Workshop, EMAS 2016, Revised, Selected, and Invited Papers*, volume 10093 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2016.

[24] P. Fraigniaud, S. Rajsbaum, M. Roy, and C. Travers. The opinion number of set-agreement. In M. K. Aguilera, L. Querzoni, and M. Shapiro, editors, *Principles of Distributed Systems: 18th International Conference, OPODIS 2014. Proceedings*, pages 155–170. Springer, 2014.

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

[26] G. Gottlob. On minimal constraint networks. *Artif. Intell.*, 191-192:42–60, 2012.

[27] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[28] P. B. Ladkin and S. Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[29] I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction-and process-oriented choreographies. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 323–332. IEEE, 2008.

[30] I. Lanese, F. Montesi, and G. Zavattaro. Amending choreographies. In A. Ravara and J. Silva, editors, *Automated Specification and Verification of Web Systems, 9th International Workshop, WWV 2013. Proceedings*, volume 123 of *EPTCS*, pages 34–48, 2013.

[31] J. A. McCarthy and S. Krishnamurthi. Cryptographic protocol explication and end-point projection. In S. Jajodia and J. López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 533–547. Springer, 2008.

[32] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

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

[34] M. P. Singh. Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language. In L. Sonenberg, P. Stone, K. Tumer, and P. Yolum, editors, *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Volume 1-3*, pages 491–498. IFAAMAS, 2011.

[35] M. P. Singh. LoST: Local state transfer - an architectural style for the distributed enactment of business protocols. In *IEEE International Conference on Web Services, ICWS 2011*, pages 57–64. IEEE Computer Society, 2011.

[36] M. P. Singh. Semantics and verification of information-based protocols. In W. van der Hoek, L. Padgham, V. Conitzer, and M. Winikoff, editors, *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012*, pages 1149–1156. IFAAMAS, 2012.

[37] B. Testerink, N. Bulling, and M. Dastani. Security and robustness for collaborative monitors. In V. Dignum, P. Noriega, M. Sensoy, and J. S. Sichman, editors, *Coordination, Organizations, Institutions, and Normes in Agent Systems XI - COIN 2015 International Workshops, Revised Selected Papers*, volume 9628 of *Lecture Notes in Computer Science*, pages 376–395. Springer, 2016.

[38] B. Testerink, M. Dastani, and N. Bulling. Distributed controllers for norm enforcement. In G. A. Kaminka, M. Fox, P. Bouquet, E. Hüllermeier, V. Dignum, F. Dignum, and F. van Harmelen, editors, *ECAI 2016 - 22nd European Conference on Artificial Intelligence – Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 751–759. IOS Press, 2016.