MAS-DRiVe: a Practical Approach to Decentralized Runtime Verification of Agent Interaction Protocols

Davide Ancona, Daniela Briola, Angelo Ferrando, and Viviana Mascardi

***Abstract*—We address the problem of decentralized runtime verification of interaction protocols in multiagent systems by means of MAS-DRiVe, an algorithm for partitioning a multi-agent system (MAS) into sub-MASs which can be monitored independently. Given a global interaction protocol named AIP (for "Agent Interaction Protocol") describing all the interactions which can take place in the MAS, the MAS-DRiVe algorithm extracts the interaction graph from AIP, identifies the clusters of agents which cannot be split during the decentralized monitoring as the interactions they are involved in are not independent, collapses each of those clusters into a single node in the interaction graph, and finally partitions the collapsed graph obtained so far. Although the "unsplittable agents identification" stage is still in its early design and prototyping phases and requires a better formalization and a deeper analysis, the MAS-DRiVe algorithm pipeline has been fully implemented and demonstrated on two simple MASs.**

Once the independently monitorable sub-MASs have been identified by MAS-DRiVe, the global interaction protocol AIP can be projected onto the subsets of agents belonging to each graph partition, thus obtaining local versions of AIP which can be monitored in a decentralized way.

***Index Terms*—Agent Interaction Protocol; Decentralized Runtime Verification; Decentralized Monitoring; Graph Partitioning**

I. Introduction and Motivation

Runtime verification (RV) is a software verification technique that complements formal static verification and testing. In RV, dynamic checking of the correct behavior of a system is performed by a monitor which is generated from a formal specification of the properties to be verified.

Distributed runtime verification (DRV), as described by S. Rajsbaum in his keynote speech at the SSS 2015 Symposium [25], tackles the problem of building a decentralized runtime monitor for a distributed system and involves designing a distributed algorithm that coordinates the monitors in order for them to correctly verify the dynamic behavior of the whole system.

With respect to RV, DRV techniques are more complex and less developed, since they involve designing a distributed algorithm that monitors another distributed algorithm. Nevertheless, they are gaining more and more attention, as shown by the recent Distributed Runtime Verification Workshop organized in Bertinoro in May 2016[1].

D. Ancona, A. Ferrando, and V. Mascardi are with DIBRIS, University of Genova, Italy. davide.ancona@unige.it, angelo.ferrando@dibris.unige.it, viviana.mascardi@unige.it

D. Briola is with DISCO, University of Milano-Bicocca, Italy. daniela.briola@unimib.it

[1]http://www.labri.fr/perso/travers/DRV2016/index.html.

Once the formal specification of the global pattern of events is given, however, distributing the monitoring activity can be resorted to decomposing the global specification into "sub-specifications", involving less events than the global one, which can be monitored in an independent way from each other and such that the union of their independent monitoring gives the same guarantees as the monitoring of the whole system w.r.t. the original specification.

Given a global specification of the expected system behavior S, the challenge to face is then to compute a set of n sub-specifications $S_1, \ldots, S_n$ derived from S in such a way that the following properties hold:

- decentralized monitoring against the sub-specifications $S_1, \ldots, S_n$ is equivalent to centralized monitoring against S;
- the decomposition of S into $S_1, \ldots, S_n$ is balanced according to some weighting criterion, to get, ideally, the best monitoring performances.

These sub-specifications do not necessarily induce a partition of the monitored events: indeed, it might be necessary that some monitored events belong to more than one specification, and hence must be checked by more monitors. Let us suppose for example that the formal specification of the event pattern states that when the alarm rings, Mary stops it and either she wakes up, washes, has breakfast and goes to work (then the monitoring is allowed to stop), or she continues to sleep.

The events "wake up", "wash", "have breakfast", "go to work" are independent from the event "sleep", so a good way to decentralize the monitoring activity between two monitors $m1$ and $m2$ could be to have $m1$ in charge of the "active" behavior ($m1$ would check that "when the alarm rings, Mary stops it and then either she wakes up, washes, has breakfast, goes to work – then the monitoring can stop –, or the monitoring can stop immediately") and $m2$ in charge of the lazy one ($m2$ would check that "when the alarm rings, Mary stops it and then either she continues to sleep, or the monitoring can stop immediately").

It would be convenient that the events "the alarm rings" and "Mary stops it" were monitored by both $m1$ and $m2$, in order to avoid a further monitor ensuring the correct sequence of initial events, whatever Mary's attitude towards the new day.

Due to events which may be monitored by more than one monitor, the total amount of monitored events in the decentralized setting may be larger than in the centralized case. However, decentralized monitors can be run in parallel on different machines, so the workload is more balanced than in the centralized setting. Also, sometimes a centralized approach is unfeasible, as in the motivating scenario discussed in our previous work presented at EMAS 2014 [2]. There, we supposed that the safety of a humanitarian convoy was guaranteed by unmanned aerial vehicles whose behavior is critical for the success of the mission and hence subject to runtime verification, but which might not be monitorable by a single monitor for network connectivity reasons and for failures of some computational entities. In cases like that, decentralizing the runtime verification activity is no longer an option, but the only viable solution. The possibility to exchange protocol specifications, dealt with as first class entities that can be sent and received [3], allows us to apply our approach to opportunistic networks (Wireless Sensor Networks, drone ferries, and the like) where the actual communication protocols are unknown until deployment, and mobility dynamically changes them.

Finding a set of sub-specifications S_1, …, S_n of S meeting the requirements for decentralizing the monitoring activity heavily depends on the way S has been specified. RV specifications are typically expressed using trace predicate formalisms. Many of them, such as finite state machines, regular expressions, context free grammars, linear-time temporal (LTL) logic, have been originally introduced for other aims and then exploited for RV; others, such as trace expressions [6], have been expressly devised for RV. Different formalisms may have different expressive power, which impacts on the types of patterns that can be described and on the meaning of "independent" sub-specifications.

Although the MAS-DRiVe algorithm proposed in this paper for decentralizing the monitoring activity is general enough to be adopted whatever the formalism and the kind of monitored events are, it has only been experimented for the specific formalism of trace expressions, for specific decentralized systems, that is, multiagent systems (MASs), and for specific events, namely interactions between pairs of agents. Also, as we will see later on, the "unsplittable agents identification" is almost naif and requires a refinement and a better theoretical support.

Trace expressions are an evolution of global types [5], which have been initially proposed for RV of interactions in MASs. Trace expressions are an expressive formalism based on a set of operators (including prefixing, concatenation, shuffle, union, and intersection) to denote finite and infinite traces of events. Their semantics is based on a labeled transition system defined by a simple set of rewriting rules which directly drive the behavior of monitors generated from them. They have been used to model and monitor (in a simulated setting) both fail-uncontrolled and ambient intelligence systems [4], as well as medical guidelines for remote patient monitoring [16].

In the already cited EMAS 2014 paper, we tackled the problem of projecting a global specification of an interaction protocol expressed in the global types formalism onto subsets of agents in the MAS. The projection function is a key element for moving from a global specification of the protocol to a localized one, which involves less agents and hence hopefully less interaction events. However, in our previous work we faced only one of the issues involved in distributing the RV activity, that of projecting the interaction protocol, and did not consider the problem of finding suitable partitions of agents in a MAS which provide formal guarantees that verification through projected types and decentralized monitors is equivalent to verification performed by a single centralized monitor with a single global type.

This paper addresses that problem by presenting MAS-DRiVe, an algorithm for partitioning a MAS in such a way that the identified subsystems can be safely monitored in a decentralized way. The paper is structured as follows: Section II discusses the related work, Section III introduces background knowledge, Section IV presents the design of MAS-DRiVe, Section V discusses its implementation and experiments, Section VI concludes.

II. Related Work

DRV lies at the crossroad where decentralized algorithms and formal methods meet [25]. It must cope both with the problems intrinsic to distributed systems, and with those related to formalization of properties to be verified at runtime. Furthermore, the distribution of the monitoring activity itself poses new problems and raises new challenges.

One of the most severe and most studied problems which characterize decentralized systems is the lack of a global clock and the need to tag events with a timestamp which, although local to the node where the event took place or was observed, can be compared with the timestamps of the other events to check their relative order. In this paper we assume that each monitor observes the events it is in charge of, in the same order as they actually took place in the environment. This condition is enough for our approach to work. As mentioned in the introduction, some events may be observed by more than one monitor so the condition we pose implicitly means that event ordering is always preserved in all monitors. A large body of literature addresses this problem; the two most widely used approaches are to connect the different physical clocks of the decentralized nodes to one common logical clock, by exploiting causality in communication between nodes (Lamports "happened before" relation [22]), and to synchronize the local clocks to provide a full ordering of events as proposed for example by M. Maróti et al. [23].

Formalizing the system properties in order to dynamically verify them requires that a suitable formal language is available. Among the first efforts in this direction we may cite one paper by A. Bauer, M. Leucker, and C. Schallhart [7] where the use of LTL in the context of RV was explored. When used for RV, the expressive power of LTL is reduced, because at runtime only finite traces can be checked. To provide a formal account for this limitation, a three-valued semantics for LTL, called LTL_3, has been proposed later on [8]. A third truth value "?" is introduced to specify that after a finite trace of events has been occurred, the outcome of a monitor can be inconclusive.

The generation of efficient monitors for properties specified in a variety of formal languages was firstly addressed by M. Kim et al. [21, 18] and many proposals exist for RV of object-oriented systems (jassda [11], PQL [24], JavaMOP [13], LARVA [14], SAGA [15], just to cite a few) and MASs [1, 17]. A large community of researchers works on RV and in 2001 the Runtime Verification Workshop/Conference series was initiated (http://www.runtime-verification.org), and workshops/conferences have occurred each year since then.

When moving from RV to DRV, however, the situation dramatically changes: no introductory papers and surveys on the topic exist, which demonstrates the youth of this research field. W.r.t. DRV of MASs, we were not able to find any related work except for one paper of ours [10] where the projection mechanism introduced in our previous EMAS 2014 work had been integrated with JADE [9].

III. Background

In this section we introduce the three pillars of our work: trace expressions, projection, and graph partitioning.

A. Trace Expressions

Trace expressions are a specification formalism expressly designed for RV. They are based on the notions of event and its abstraction, event type.

Events. In the following we denote by $\mathcal{E}$ a fixed universe of events. An event trace over $\mathcal{E}$ is a possibly infinite sequence of events in $\mathcal{E}$. As an example, we might have $\mathcal{E} = \{$ *msg(bob, alice, tell, m1), msg(alice, bob, tell, a1), msg(bob, carol, tell, m2), msg(carol, bob, tell, a2), msg(bob, dave, tell, m3), msg(dave, bob, tell, a3)* $\}$ where the interaction event *msg(S, R, P, C)* corresponds to the interaction between agent *S* and agent *R*, with *S* sending a message with performative *P* and content *C* to *R*. In this example, contents *m1*, *m2* and *m3* correspond to actual messages, and contents *a1*, *a2* and *a3* correspond to acknowledges of reception.

Event types. To be more general, trace expressions are built on top of event types (chosen from a set $\mathcal{ET}$), rather than of single events; an event type denotes a subset of $\mathcal{E}$. For example, if we were interested only in the type of the message content (actual message or acknowledge), we might define the two event types *Msg*= $\{$ *msg(bob, alice, tell, m1), msg(bob, carol, tell, m2), msg(bob, dave, tell, m3)* $\}$ and *Ack*= $\{$ *msg(alice, bob, tell, a1), msg(carol, bob, tell, a2), msg(dave, bob, tell, a3)* $\}$, and use them for describing interaction patterns.

Trace expressions. A trace expression τ represents a set of possibly infinite event traces, and is defined on top of the following operators (binary operators associate from left, and are listed in decreasing order of precedence, that is, the first operator 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 e matches the event type ϑ ($e \in \vartheta$), and the remaining part is a trace of τ. For example, if our communication protocol just required that an interaction from the set *Msg* must take place, no matter which one, and then an interaction from *Ack* must occur, no matter which one, we could express it as *Msg*:*Ack*:ϵ.
- $\tau_1{\cdot}\tau_2$ (*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 in τ_1 with the traces in τ_2.

To support recursion without introducing an explicit construct, trace expressions are regular (a.k.a. rational or cyclic) terms. A regular term can be represented by a finite set of syntactic equations, as happens, for instance, in most modern Prolog implementations where unification supports cyclic terms. A trace expression τ is contractive if all its infinite paths from the root contain the prefix operator.

From the operators above we can derive the filter operator, useful for making trace expressions more compact and readable. The expression $\vartheta{\gg}\tau$ denotes the set of all traces contained in τ, when deprived of all events that do not match ϑ. Assuming that event types are closed by complementation, the expression above is a convenient syntactic shortcut for $T|\tau$, where $T = \epsilon{\vee}\overline{\vartheta}{:}T$, and $\overline{\vartheta}$ is the complement event type of ϑ, that is, $[\![\overline{\vartheta}]\!] = \mathcal{E} \setminus [\![\vartheta]\!]$.

In the context of RV, where the three-valued semantics is considered for LTL, trace expressions are strictly more expressive than LTL [6]: every LTL formula can be encoded into a trace expression with an equivalent three-valued semantics, whereas the opposite property does not hold, since trace expressions are also able to specify all context-free languages, and also some non context-free ones.

Although trace expressions can define protocols involving events of any kind, in the sequel we will limit our investigation to interaction events like *msg(S, R, P, C)* and interaction types that we will identify with α.

B. Projection

The projection function was first introduced in [2]. Given the finite set AGS of all the agents that could play a role in the MAS and an interaction type α, *senders*(α) is the set of all the agents in AGS that could play the role of sender in actual interactions having type α, and *receivers*(α) is the set of all the agents in AGS that could play the role of receiver in interactions of type α. The *involves* predicate holds on one interaction type α and one set of agents Ags, *involves*(α, Ags), iff $($*senders*$(\alpha) \cap Ags \neq \emptyset) \vee ($*receivers*$(\alpha) \cap Ags \neq \emptyset)$.

Projection can be described as a function $\Pi : \mathcal{T} \times \mathcal{P}(AGS) \rightarrow \mathcal{T}$ where $\mathcal{T}$ is the set of trace expressions. Π is driven by the syntax of the trace expression to project; since Π is defined on cyclic terms, the simplest way to define it would be by coinduction[2] as follows:

(i) $\Pi(\epsilon, Ags) = \epsilon$

[2]Coinduction is a technique for defining and proving properties of systems of concurrent interacting objects. It is the mathematical dual to structural induction. Coinductively defined types are typically infinite data structures, such as streams. See http://cseweb.ucsd.edu/groups/tatami/handdemos/doc/coind.htm for a more technical formalization.

(ii) $\Pi(\alpha : \tau, Ags) = \alpha : \Pi(\tau, Ags)$ if *involves*(α, Ags)

(iii) $\Pi(\alpha : \tau, Ags) = \Pi(\tau, Ags)$ if $\neg$*involves*(α, Ags)

(iv) $\Pi(\tau' \; op \; \tau'', Ags) = \Pi(\tau', Ags) \; op \; \Pi(\tau'', Ags)$, where $op \in \{\cdot, \wedge, \vee, |\}$.

This definition works properly only on all non cyclic terms and on some, but not all, cyclic terms. To guarantee that the projection function always returns a contractive type and that the correct coinductive definition is implemented, it is necessary to keep track of all types visited by Π along a path; each type is associated with its depth in the path, and with a fresh variable which will be unified with the corresponding computed projection. During the visit, the depth *DeepestSeq* of the deepest visited sequence operator is kept. If a type τ has been already visited, then a cycle is detected: if its depth is less than *DeepestSeq* then the cycle contains an occurrence of the sequence constructor, therefore the projected type associated with τ is contractive and, hence, is returned; otherwise, the projection would not be contractive, therefore ϵ is returned.

C. Graph Partitioning

The graph partitioning problem (GPP) is defined in the following way [12]: given a number $k \in \mathbb{N}_{>1}$ and an undirected graph $G = (V, E)$ with non-negative node and edge weights, GPP asks for a partition of V, that is, with blocks of nodes $(V_1, ..., V_k)$ such that

1) $V_1 \cup V_2 \cup ... \cup V_k = V$
2) $V_i \cap V_j = \emptyset \quad \forall i \neq j$

A balance constraint may be required, demanding that all blocks have about equal weights. To be more precise, a good partition is the one where the sum of the node weights in each V_i is "about the same" and the sum of all edge weights of edges connecting all different pairs V_i and V_j is minimized. Applications of graph partitioning include parallel processing, road networks, image processing, VLSI design, social networks, and bioinformatics. Among the many existing tools for graph partitioning we opted for using METIS [20], a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices (http://glaros.dtc.umn.edu/gkhome/views/metis, accessed on June 2016). The choice of METIS was due to the efficiency of the multi-level Kernighan/Lin approach it builds upon [19] and to the ease of its installation and usage.

IV. MAS-DRiVe: Design

Given an interaction protocol expressed in some suitable formalism, the MAS-DRiVe algorithm for partitioning a MAS in such a way that the identified subsystems can be safely monitored in a decentralized way is the following:

1) extract the interaction graph IG from the interaction protocol;
2) identify the sets of agents which cannot be split during the decentralized monitoring (*unsplittable agents*), since the interactions they are involved in are not independent;
3) compute the collapsed interaction graph CIG by collapsing each of those sets into a single node with weight equal to the cardinality of the collapsed set, and by computing the collapsed edges consistently; for simplicity, all edges in the CIG have weight 1, but the approach works also in case edge weights are different: a higher edge weight would model a preferential communication channel;
4) partition the new graph obtained so far using some suitable graph partition method;
5) project the interaction protocol onto these agents' partitions.

The algorithm is explained by means of two running examples introduced in [2].

A. Examples

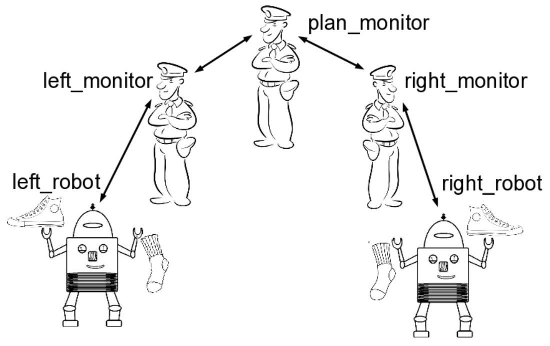

Fig. 1. The "Socks and Shoes" MAS

a) Socks and Shoes (SaS): Let us consider the simple scenario illustrated in Figure 1 where two robots (right and left), two monitors (right and left) associated with each robot, and a plan monitor which supervises them, must reach the goal to put the right and left socks and shoes in the correct way (socks first!). As robots are autonomous, they could perform the two actions in the wrong order: monitors are there to ensure that wrong actions are immediately rolled back. Robots communicate their actions to their corresponding monitors, which, in turn, notify the plan monitor when the robots accomplish their goal. The interaction types involving the right robot and the right node monitor are defined as *msg(right_robot, right_node_monitor, tell, put_sock)* $\in$ *put_right_sock*; *msg(right_robot, right_node_monitor, tell, put_shoe)* $\in$ *put_right_shoe*; *msg(right_robot, right_node_monitor, tell, removed_shoe)* $\in$ *removed_right_shoe*; *msg(right_node_monitor, right_robot, tell, oblige_remove_shoe)* $\in$ *oblige_remove_right_shoe*; *msg(right_node_monitor, plan_monitor, tell, ok)* $\in$ *ok_right*. Those for the left robot and left node monitor are symmetrical.

The right robot (*RIGHT* branch in the protocol description below) can start by putting the sock, which is the correct action to do, or ($\vee$ operator) by putting the shoe, which requires a recovery by the right robot monitor and looping back to the *RIGHT* branch. The left robot has the same behavior (*LEFT* branch in the protocol description). The *SaS* protocol is described by the shuffle of the actions of the right and left robots and monitors (*RIGHT* | *LEFT*):

$$\begin{aligned}
\mathit{RIGHT} &= (\mathit{put\_right\_sock}{:}\mathit{put\_right\_shoe}{:}\mathit{ok\_right}{:}\epsilon)\vee \\
&\quad (\mathit{put\_right\_shoe}{:}\mathit{oblige\_remove\_right\_shoe}{:} \\
&\quad \mathit{removed\_right\_shoe}{:}\mathit{RIGHT}) \\
\mathit{LEFT} &= (\mathit{put\_left\_sock}{:}\mathit{put\_left\_shoe}{:}\mathit{ok\_left}{:}\epsilon)\vee \\
&\quad (\mathit{put\_left\_shoe}{:}\mathit{oblige\_remove\_left\_shoe}{:} \\
&\quad \mathit{removed\_left\_shoe}{:}\mathit{LEFT}) \\
\mathit{SaS} &= \mathit{RIGHT}|\mathit{LEFT}.
\end{aligned}$$

b) Alternating Bit Protocol with four participants (ABP3): As a second example we consider the Alternating Bit Protocol ABP.

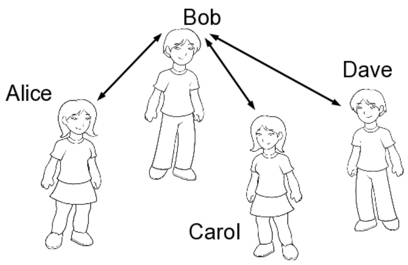

Fig. 2. The ABP MAS with one "manager" (Bob) and three participants.

We consider the instance of ABP that we name *ABP3*, where six different interactions may occur: Bob sends *m1* to Alice (interaction type msg_1), Alice sends *a1* to Bob (interaction type ack_1), Bob sends *m2* to Carol (interaction type msg_2), Carol sends *a2* to Bob (interaction type ack_2), Bob sends *m3* to Dave (interaction type msg_3), Dave sends *a3* to Bob (interaction type ack_3). The *ABP3* is an infinite iteration, where the following constraints have to be satisfied for all occurrences of the sending actions:

1. The n-th occurrence of an interaction of type msg_1 must precede the n-th occurrence of an interaction of type msg_2 which in turn must precede the n-th occurrence of an interaction of type msg_3 which in turn must precede the $n+1$-th occurrence of an interaction of type msg_1.

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

ABP3 can be defined by the following set of equations:

$$\begin{aligned}
\mathit{ABP3} &= (msg \gg MM)\wedge(msg\_ack(1)\gg MA_1)\wedge \\
&\quad (msg\_ack(2)\gg MA_2)\wedge(msg\_ack(3)\gg MA_3) \\
MM &= msg_1{:}msg_2{:}msg_3{:}MM \\
MA_1 &= msg_1{:}ack_1{:}MA_1 \\
MA_2 &= msg_2{:}ack_2{:}MA_2 \\
MA_3 &= msg_3{:}ack_3{:}MA_2
\end{aligned}$$

where $msg\_ack(i)$, $i \in \{1,2,3\}$, and msg denote the event types s.t. $[\![msg\_ack(i)]\!] = [\![msg_i]\!] \cup [\![ack_i]\!]$, $i \in \{1,2,3\}$, and $[\![msg]\!] = [\![msg_1]\!] \cup [\![msg_2]\!] \cup [\![msg_3]\!]$.

The trace expression defined by MM corresponds to the first constraint informally stated above, while MA_1, MA_2, and MA_3 formalize the second constraint. The main trace expression *ABP3* can be easily read as follows: if an event has type msg_1 or msg_2 or msg_3, then it must verify MM, and if an event has type msg_1 or ack_1, then it must verify MA_1, and if an event has type msg_2 or ack_2, then it must verify MA_2, and if an event has type msg_3 or ack_3, then it must verify MA_3.

B. Algorithm Steps

As already introduced in the beginning of this section, the MAS-DRiVe algorithm consists of five steps.

1) Interaction graph extraction. The interaction graph IG of the agents interaction protocol AIP expressed using formalism $\mathcal{F}$ is an undirected and unweighted graph with one node A_i for each agent A_i involved in AIP. An edge between A_i and A_j exists iff an interaction between A_i and A_j is foreseen by AIP. By "interaction between A_i and A_j" we mean that either A_i sends a message to A_j, or viceversa.

As an example, the nodes extracted from the *SaS* protocol are *right_robot*, *right_node_monitor*, *plan_monitor*, *left_robot*, *left_node_monitor* and the edges are (*right_robot, right_node_monitor*), (*right_node_monitor, plan_monitor*), (*left_robot, left_node_monitor*), (*left_node_monitor, plan_monitor*).

The interaction graph algorithm explores the trace expression to find pairs of agents involved in the interaction types. Since the trace expression may be cyclic, loops must be detected and dealt with properly. This goal is achieved in a simple way: trace expressions met during exploration are recorded, and when the exploration comes back to an already recorded trace, it stops.

To identify the sets of unsplittable agents during the second step of the algorithm, during this stage an auxiliary directed multigraph IG_{ext} is generated, where each edge connecting two nodes is labeled with a string that encodes the path followed to reach that interaction in the protocol. The nodes and edges in IG_{ext} are obtained by means of the *extract(ID, AlreadyMetTrExp, Str, Level, τ)* function where

– *ID* is the protocol identifier;

– *AlreadyMetTrExp* is the set of trace expressions which were already met during the protocol analysis, to take care of cycles;

– *Str* is the current path encoding;

– *Level* is an encoding of the current outermost operator level and position (left or right) in the trace expression parse tree;

– τ is the trace expression currently under consideration.

extract(ID, AlreadyMetTrExp, Str, Level, τ) is defined by cases:

1) if *TrExpr* belongs to *AlreadyMetTrExp*, then return;
2) if *TrExpr* is ϵ, then return;
3) if τ is $\alpha\ uop\ \tau_1$, where uop is an operator in $\{:, \gg\}$, then
   a) *NewStr = Str • (uop, Level+1)*
   b) extract all the possible pairs of senders and receivers (S, R) such that S and R are involved in α
   c) for each of such pairs, generate and assert *edge(ID, S, R, NewStr)*
   d) *NewAlreadyMetTrExp* = $\{\tau\} \bigcup$ *AlreadyMetTrExp*

```
        e) call extract(ID, NewAlreadyMetTrExp, NewStr, Level+1, $\tau_1$)
    4) if $\tau$ is $\tau_1 \wedge \tau_2$, then
        a) NewStr = Str $\bullet$ ($\wedge$, Level+1)
        b) NewAlreadyMetTrExp = $\{\tau\} \bigcup$ AlreadyMetTrExp
        c) call extract(ID, NewAlreadyMetTrExp, NewStr, Level+1, $\tau_1$)
        d) call extract(ID, NewAlreadyMetTrExp, NewStr, Level+1, $\tau_2$)
    5) if $\tau$ is $\tau_1$ bop $\tau_2$, where bop is an operator in $\{\cdot, \vee, |\}$, then
        a) NewStr1 = Str $\bullet$ (bop, Level+1)
        b) NewStr2 = Str $\bullet$ (bop, 2*Level+1)
        c) NewAlreadyMetTrExp = $\{\tau\} \bigcup$ AlreadyMetTrExp
        d) call extract(ID, NewAlreadyMetTrExp, NewStr1, Level+1, $\tau_1$)
        e) call extract(ID, NewAlreadyMetTrExp, NewStr2, 2*Level+1, $\tau_2$)
```

Given a protocol identified by *ID* and represented by the trace expression τ, *extract(ID,* $\{\}$, Λ, *1,* τ*)* generates and asserts all the edges in IG_{ext}. The generation of IG_{ext} nodes is straightforward. The idea behind this function is that edges whose labels share the same prefix have a strong dependency as they belong to the same intersection. In fact, the only case where labels of edges share the same prefix is that of $\wedge$ (case 4), where the same string and level are passed to the recursive calls. If the operator is not an intersection (case 5), the strings and levels passed to the recursive calls are different, meaning that the two operands are independent from one another and so will be the edges generated during their exploration.

For example, the edges in the auxiliary IG_{ext} graph extracted from the *ABP3* protocol are:

(*bob*, *alice*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5)))
(*bob*, *carol*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5)))
(*bob*, *dave*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5)))
(*bob*, *alice*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5) (:, 6)))
(*bob*, *carol*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5) (:, 6) (:, 7)))
(*bob*, *dave*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5) (:, 6) (:, 7) (:, 8)))
(*alice*, *bob*, (($\wedge$, 2) ($\wedge$, 3) ($\wedge$, 4) ($\gg$, 5) (:, 6) (:, 7)))
(*bob*, *carol*, (($\wedge$, 2) ($\wedge$, 3) ($\gg$, 4)))
(*carol*, *bob*, (($\wedge$, 2) ($\wedge$, 3) ($\gg$, 4)))
(*bob*, *carol*, (($\wedge$, 2) ($\wedge$, 3) ($\gg$, 4) (:, 5)))
(*carol*, *bob*, (($\wedge$, 2) ($\wedge$, 3) ($\gg$, 4) (:, 5) (:, 6)))
(*bob*, *dave*, (($\wedge$, 2) ($\gg$, 3)))
(*dave*, *bob*, (($\wedge$, 2) ($\gg$, 3)))
(*bob*, *dave*, (($\wedge$, 2) ($\gg$, 3) (:, 4)))
(*dave*, *bob*, (($\wedge$, 2) ($\gg$, 3) (:, 4) (:, 5)))

The IG nodes are *alice*, *bob*, *dave*, *carol* and the edges are (*alice*, *bob*), (*dave*, *bob*), (*carol*, *bob*), which are extracted in a trivial way from IG_{ext}.

2) Identification of unsplittable agents. Not all the agents can be monitored independently from one another, as some interaction patterns that must be respected by a set of agents seen as a whole, could be lost when looking at subsets of the agents.

As an example, *alice*, *bob*, *carol* and *dave* in the *ABP3* represent an unsplittable set of agents. In fact, if we monitored interactions involving *alice* and *bob* only, we could find that they respect the second constraint of the protocol, but we could never ensure that the first constraint – which also depends on the interactions of *bob* with *carol* and *dave* – is respected as well. The notions of dependence and independence are related to the protocol but also to the formalism used for modeling it, so no general rule for this step can be devised. When the interaction formalism $\mathcal{F}$ is that of trace expressions, however, dependencies among interactions are naturally modeled using the intersection operator. Thus, although it may be an over-cautious approach, when using trace expressions we define "unsplittable" the agents involved in interactions connected by an intersection operator.

The algorithm for performing the identification of unsplittable agents looks at the labels in IG_{ext} and creates sets of agents involved in interactions whose label has the same prefix before the first $\wedge$ operator in the label, if any. For example, in *SaS* no label contains $\wedge$ and hence the set of unsplittable agents is empty, whereas in *ABP3* all the edge labels share the ($\wedge$, 2) prefix, meaning that all the interactions in the protocol belong to branches connected by the same intersection operator. As a consequence, all the agents involved in the *ABP3* protocol interactions belong to the same set of unsplittable agents, which coincides with all the agents in the MAS.

3) Graph collapse. In this stage, we transform the unweighted undirected interaction graph IG into a weighted undirected graph CIG, obtained by collapsing each node in IG corresponding to an agent belonging to unsplittable set U, into a single node with label U and weight equal to U cardinality. Nodes in CIG are labeled with sets of agents, which can be the singleton set for agents belonging to no unsplittable set. Edges exiting from (resp. entering into) a node are all those which, in IG, exited from (resp. entered into) one of the nodes in the label.

Considering the running examples above, the weighted nodes in the *SaS* CIG are *(*$\{$*right_robot*$\}$*,1)* *(*$\{$*right_node_monitor*$\}$*,1)* *(*$\{$*plan_monitor*$\}$*,1)* *(*$\{$*left_robot*$\}$*,1)* *(*$\{$*left_node_monitor*$\}$*,1)* namely, no collapse took place, while the only weighted node in the *ABP3* CIG is *(*$\{$*alice,bob,carol,dave*$\}$*,4)* labeled with the set of unsplittable agents.

The edges in the *SaS* CIG are the same as those in the *SaS* IG, while there are no edges in the *ABP3* CIG due to the presence of only one node.

4) Graph partitioning. This stage of the algorithm consists in partitioning the CIG graph obtained by the collapse stage. The number of expected partitions is given as an input by the user and must be grater than one. Any suitable partitioning algorithm could be used during this stage. We will discuss our implementative choice in Section V.

The partition of the original, not collapsed graph IG, is trivially derived by that of CIG by making the union of the agents in the labels of the nodes in each CIG partition.

By using a graph partitioning algorithm where the sum of the node weights in each partition is about the same and the number of edges connecting nodes in two different partitions is minimized (by associating the unit weight with each edge), we can obtain the following partition for the agents involved in the *SaS* protocol: {{*left_node_monitor, left_robot*}, {*plan_monitor, right_node_monitor, right_robot*} } .

The agents involved in the *ABP3* protocol cannot be partitioned because the collapsed graph consists of only one node.

5) Projection. Once the agents belonging to the same partition have been devised, the projection algorithm presented in Section III-B can be used to project the global interaction protocol onto each of them.

V. Implementation and Experiments

The MAS-DRiVe algorithm is fully implemented in SWI-Prolog (http://www.swi-prolog.org/, accessed on June 2016), apart from the graph partitioning stage which exploits the METIS tool described in Section III. The extraction of the IG from the protocol, the identification of the unsplittable agents sets, and the generation of the collapsed graph CIG are performed by Prolog predicates. After them, a predicate for encoding CIG into a format suitable for METIS and saving it into a file is called. The command line for running METIS on that file is launched from inside Prolog, and the result is read and decoded.

In order to run the algorithm, the user must call the `partition/3` predicate with three arguments: the identifier of the agent interaction protocol to distribute, the number of partitions to be obtained, and the name of the file where Prolog will write the result of the partition. For example, given that `abp3` identifies the *ABP3* protocol, and that the trace expression associated with this identifier has been read into the Prolog knowledge base, the user should call `partition(abp3, 2, './abp3Out.txt')` obtaining the following content for the `abp3Out.txt` file:

```
Unsplittable list:
[((/\),2)],[alice,bob,carol,dave]

Partition failed

Nodes
node(abp3,bob)
node(abp3,alice)
node(abp3,carol)
node(abp3,dave)

Extended edges
edge(abp3,bob,alice,[ ((/\),2), ((/\),3),
                      ((/\),4), ((>>),5)])
edge(abp3,bob,carol,[ ((/\),2), ((/\),3),
                      ((/\),4), ((>>),5)])
....

Collapsed nodes
cnode(abp3,[alice,bob,carol,dave],4)
```

The content of the `socksOut.txt` file that we obtain by calling `partition(socks, 2, './socksOut.txt')` is:

```
Unsplittable list: []

The trace expressions
@((S_1|S_2),
[S_1=
(put_right_sock:put_right_shoe:ok_right:lambda)\/
(put_right_shoe:oblige_remove_right_shoe:
                        removed_right_shoe:S_1),
S_2=
(put_left_sock:put_left_shoe:ok_left:lambda)\/
(put_left_shoe:oblige_remove_left_shoe:
                       removed_left_shoe:S_2)])

can be partitioned into

{[left_node_monitor],[left_robot]}
{[plan_monitor],[right_node_monitor],[right_robot]}


Nodes
node(socks,right_robot)
node(socks,right_node_monitor)
node(socks,plan_monitor)
node(socks,left_robot)
node(socks,left_node_monitor)

Extended edges
edge(socks,right_robot,right_node_monitor,
[((|),2), ((\/),3), ((:),4)])

edge(socks,right_robot,right_node_monitor,
[((|),2), ((\/),3), ((:),4), ((:),5)])
........

Collapsed nodes
cnode(socks,[right_robot],1)
cnode(socks,[right_node_monitor],1)
cnode(socks,[plan_monitor],1)
cnode(socks,[left_robot],1)
cnode(socks,[left_node_monitor],1)

Collapsed edges
cedge(socks,[right_robot],[right_node_monitor])
cedge(socks,[right_node_monitor],[plan_monitor])
cedge(socks,[left_robot],[left_node_monitor])
cedge(socks,[left_node_monitor],[plan_monitor])
```

We run experiments with protocols that involve distinct groups of agents each following the ABP3 protocol. For example, in the *DoubleABP3*, *alice*, *bob*, *carol* and *dave* follow the *ABP3* protocol, *alice2*, *bob2*, *carol2* and *dave2* follow the *ABP3* protocol as well, and both *bob* and *bob2* interact with *boss*. Since *alice*, *bob*, *carol* and *dave* have no interactions with *alice2*, *bob2*, *carol2* and *dave2*, the two groups are independent from each other and can be safely partitioned for the purpose of decentralized runtime verification. In such a situation, the partition computed by MAS-DRiVe is the following:

```
Unsplittable list:
[ ((*),3), ((\/),7), ((:),8), ((:),9), ((/\),10)],
                          [alice2,bob2,carol2,dave2]
[ ((*),3), ((\/),4), ((:),5), ((:),6), ((/\),7)],
                               [alice,bob,carol,dave]

The trace expressions
@((bobasks:lambda|bob2asks:lambda)*
((okbob:nobob2:msg3>>S_1/\msg_ack(1)>>S_2/\
msg_ack(2)>>S_3/\msg_ack(3)>>S_4)\/
(okbob2:nobob:mmsg3>>S_5/\mmsg_ack(1)>>S_6/\
mmsg_ack(2)>>S_7/\mmsg_ack(3)>>S_8)),
[S_1=m1:m2:m3:S_1,S_2=m1:a1:S_2,
S_3=m2:a2:S_3,S_4=m3:a3:S_4,S_5=mm1:mm2:mm3:S_5,
S_6=mm1:aa1:S_6,S_7=mm2:aa2:S_7,S_8=mm3:aa3:S_8])
```

```
can be partitioned into

{[boss],[alice2,bob2,carol2,dave2]}
{[alice,bob,carol,dave]}
```

The algorithm correctly recognizes that *alice*, *bob*, *carol* and *dave* must be monitored all together, and that the same holds for *alice2*, *bob2*, *carol2* and *dave2*. No other constraints on the partition are found. We got the expected correct results also with the *TripleABP3* protocol, with three distinct groups of agents following the *ABP3*.

The code of the MAS-DRiVe algorithm can be downloaded from http://www.disi.unige.it/person/MascardiV/Software/masdrive.html. It requires SWI Prolog and the METIS software installed and accessible via the command line from everywhere. We tested it on Mageia Linux.

VI. Conclusions and Future Work

This paper addresses the problem of how to partition a MAS into agents' subsets that can be verified at runtime, independently from one another, in such a way that the results obtained by the decentralized runtime verification are the same as those obtained by monitoring the whole MAS in a centralized way.

We designed and implemented the MAS-DRiVe algorithm, whose results are consistent with those discussed in [2]. In that paper, the feasibility of distributing the runtime monitoring over user-defined sets of agents was empirically validated by checking the traces of events compliant with the decentralized subprotocols up of a given length, and verifying whether they were compliant with the original global protocol as well. That approach gave a semi-decidable information: if all traces up to a given length l were found to be compliant with the protocol, this did not ensure that problems could not arise with traces of length $l + 1$. With MAS-DRiVe we propose a constructive way to partition agents in the MAS, rather than letting the user decide how to partition them, and we overcome the problems due to the empirical validation approach by proposing partitions that are "safe" w.r.t. the trace expression operators. In particular, we assume that agents involved in an intersection can never be partitioned. This is an over-cautious approach, but since intersection is the construct used to state constraints across different independent branches of the protocol, we assume that those branches cannot be monitored independently.

In its current version, then, the MAS can not get split if no semi-independent parts of the system can get identified. Although this is a limitation of our approach, this could have an interesting side effect that we aim to explore in the close future, namely the possibility to come up with suggestions for MAS designers in terms of good practices for better sub-systems encapsulation based on interaction protocol analysis. Such suggestions could be applied to distributed systems in general, and not only to MASs, and could prove useful in those approaches where there is an embedded notion of some sort of encapsulation.

As far as graph partitioning is concerned, we point out that the obtained partitions do not correspond to partitions of events. For example, an interaction between the *plan_monitor* and the *left_node_monitor*, corresponding to a (*plan_monitor*, *left_node_monitor*) edge in the graph, will be monitored by both the monitor in charge of $\{$*left_node_monitor, left_robot*$\}$ and that in charge of $\{$*plan_monitor, right_node_monitor, right_robot*$\}$, as it involves agents in both of them. This redundancy cannot be avoided, but can be kept as small as possible by selecting a partitioning algorithm that minimizes the number of edges across partitions. Graph partitioning is another area of improvement of our work: strategies could be employed to automatically detect the optimal number of monitors to deploy, or at least guidelines may be provided in the future.

Finally, as part of our future work, we plan to make an exhaustive experimentation of MAS-DRiVe on existing complex protocols. METIS can easily partition graphs with hundreds of nodes so we will model protocols involving hundreds of agents and test the scalability of the approach. Studying its computational properties from a theoretical viewpoint is on our agenda as well.

Acknowledgments

We are grateful to the anonymous reviewers for their useful remarks and constructive comments.

References

[1] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. The *S*CIFF abductive proof-procedure. In *AI*IA*, pages 135–147, 2005.

[2] D. Ancona, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert. Efficient verification of MASs with projections. In *EMAS 2014. Revised Selected Papers*, volume 8758 of *LNCS*, pages 246–270. Springer, 2014.

[3] 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, *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015*, pages 1019–1029. ACM, 2015.

[4] 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.

[5] 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. Vasconcelos, editors, *DALT 2012. Revised Selected Papers*, volume 7784 of *LNCS*. Springer, 2013.

[6] D. Ancona, A. Ferrando, and V. Mascardi. Comparing trace expressions and linear temporal logic for runtime verification. In E. Ábrahám, M. M. Bonsangue, and E. B. Johnsen, editors, *Theory and Practice of Formal Methods*, volume 9660 of *LNCS*, pages 47–64. Springer, 2016.

[7] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. and Comput.*, 20(3):651–674, June 2010.

[8] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, Sept. 2011.

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

[10] 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, *IDC 2014*, volume 570 of *Studies in Computational Intelligence*, pages 81–91. Springer, 2014.

[11] M. Brörkens and M. Möller. Dynamic event generation for runtime checking using the JDI. *Electr. Notes Theor. Comput. Sci.*, 70(4):21–35, 2002.

[12] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *LNCS*. Springer, 2015 (in press).

[13] F. Chen and G. Rosu. Mop: an efficient and generic runtime verification framework. In *OOPSLA 2007*, pages 569–588, 2007.

[14] C. Colombo, G. J. Pace, and G. Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM 2009*, pages 33–37, 2009.

[15] F. S. de Boer and C. P. T. de Gouw. Combining Monitoring With Run-Time Assertion Checking. In *SFM 14*, pages 217 – 262. Springer, 2014.

[16] A. Ferrando, V. Mascardi, and D. Ancona. Monitoring patients with hypoglycemia using self-adaptive protocol-driven agents: a case study. In *Workshop Proceedings of EMAS 2016*, 2016.

[17] L. Giordano, A. Martelli, and C. Schwind. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic*, 5(2):214 – 234, 2007.

[18] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In J. Katoen and P. Stevens, editors, *TACAS 2002*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.

[19] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[20] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.

[21] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs. *Electr. Notes Theor. Comput. Sci.*, 55(2):218–235, 2001.

[22] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[23] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *SenSys '04*, pages 39–49. ACM, 2004.

[24] M. C. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA 2005*, pages 365–383, 2005.

[25] S. Rajsbaum. Distributed runtime verification – where combinatorics, fault-tolerance and formal methods meet. Keynote Talk at the SSS 2015, August 2015, 2015.