Efficient Verification of MASs with Projections

Davide Ancona, Daniela Briola, Amal El Fallah Seghrouchni,
Viviana Mascardi, and Patrick Taillibert

[1] DIBRIS, University of Genova, Italy
{Davide.Ancona,Daniela.Briola,Viviana.Mascardi}@unige.it
[2] LIP6, University Pierre and Marie Curie, Paris, France
{Amal.Elfallah,Patrick.Taillibert}@lip6.fr

Abstract. Constrained global types are a powerful means to represent agent interaction protocols. In our recent research we used them to represent complex protocols in a very compact way, and we exploited them to dynamically verify actual agents' interactions with respect to different protocols in both Jason and JADE. The main drawback of our previous approach is the full centralization of the monitoring activity, which is delegated to a unique monitor agent in charge of verifying that the messages exchanged among all the agents are compliant with the protocol. This approach works well for MASs with few agents, but could become unsuitable in communication-intensive and highly-distributed MASs where hundreds of agents should be monitored.

In this paper we define an algorithm for projecting a constrained global type onto a set of agents *Ags*, by restricting it to the interactions involving agents in *Ags*, so that the outcome of the algorithm is another constrained global type where interactions involve only agents in *Ags*. The projection mechanism is the first step towards distributing the monitoring activity, making it safer and more efficient: the compliance of a MAS to a protocol could be dynamically verified by suitably partitioning the agents of the MAS into small sets of agents, and by assigning to each partition *Ags* a local monitor agent which checks all interactions involving *Ags* against the projected constrained global type.

Although the projection of well formed constrained global types can be always performed, the resulting projected protocol does not always model all the constraints as the original one. We describe a generate and test algorithm that provides hints on the correctness of the protocol distribution, leaving for further investigation the formal characterization of which protocols can be distributed onto which agents' subsets.

Keywords: Constrained Global Type, Projection, Dynamic Verification, Agent Interaction Protocol.

1 Introduction and Motivation

Distributed monitoring of agent interaction protocols is interesting for various reasons. First, the distribution of monitoring reduces the bottleneck issue due to the potentially high number of communications between the central monitor

F. Dalpiaz et al. (Eds.): EMAS 2014, LNAI 8758, pp. 246–270, 2014.

and the agents of the system. Consequently, the communications are localized according to the distribution topology (how many local monitors are available and where they are localized in the system), improving the efficiency of the monitoring. As usual, distribution increases the robustness of the whole system and prevents for a breakdown, crash or failure of the system. In particular, in the context of distributed environments, having a robust monitoring system requires to distribute the monitoring on several agents which ensure their prompt reaction to events. In addition, the distributed approach is more suitable than the centralized one for asynchronous and/or distributed contexts. Hence, we can mention at least three classes of applications where the distribution of monitoring is relevant.

1. MASs dealing with huge number of agents, for example applications in the context of supervising networks (e.g. [28]). The distribution becomes mandatory to deal with the complexity of the system and to guarantee its scalability.

2. Distributed MASs dealing with distributed agents because of the intrinsic geographical distribution of the system. This often happens in the context of industrial projects.

3. Pervasive MASs: in ambient intelligent systems for instance, agents are mobile (they move from one locality to another one) and their communication depends on their location. In such open environments, agents enter and leave the system and this requires a distributed monitoring of communication (e.g. local registration, etc.).

Usually, in systems related to the above three classes of applications, an overlay of agents is deployed above the real system. Agents are distributed over the system according to the topology distribution which has to satisfy several criteria (logical, physical or temporal, etc.) of communication in order to meet the target application requirements. The induced topology leads the agents to communicate with their local monitor or with their neighboring agents in order to exchange information.

In order to distribute the monitoring activity, the first step to face is to design and implement an algorithm for *projecting the protocol specification onto subsets of agents*, and then allow interactions taking place within these subsets to be monitored by local monitors. This step is the main subject of this paper.

Automatically identifying these subsets of agents in order to guarantee that the distributed monitoring behaves like the centralized one is the second step to face. The current solution to this issue is a generate and test algorithm which may detect the impossibility to distribute the monitoring activity, without however guaranteeing the possibility to distribute it. We leave for further investigation the problem of finding suitable partitions of agents in a MAS which provide formal guarantees that verification through projected types and distributed agents is equivalent to verification performed by a single centralized monitor with a "centralized" global type.

A third interesting issue concerns dynamic redistribution of monitoring agents; even if not explored in this work, projected types could be recomputed dynamically to balance the load among local monitors depending on the currently available

resources, and according to some "meta-protocol". Self-adaptation of local monitors along the lines of [13] raises similar issues as dynamic redistribution.

We exploit the formalism of constrained global types [2] for specifying and dynamically verifying agent interaction protocols. In our recent research we demonstrated that they can be used to represent complex protocols in a very compact way, and we exploited them to detect deviations from the protocol in both Jason[1] [3] and JADE[2] [8]. Extensions of the original formalism with attributes have been described [20] and exploited to model a complex, real protocol in the railway domain [21]. This paper shows how a constrained global type can be projected onto a set of agents *Ags*, obtaining another constrained global type which contains only interactions involving agents in *Ags*. Although the projection of a well formed global type is always possible, this does not mean that it is always meaningful: as an example, the Alternating Bit Protocol (ABP) that will be introduced later on in this paper can be projected onto any subset of agents in the MAS, but needs to be monitored in a centralized way to verify all its constraints. Our generate and test algorithm detects the impossibility to distribute the monitoring of the ABP, hence providing a useful, although partial, support to the protocol and MAS developers.

The paper is organized in the following way: the sequel of this section describes one motivating scenario for our research; Section 2 overviews the state of the art in runtime monitoring of distributed systems; Section 3 gives the technical background needed for presenting the projection algorithm in Section 4, Section 5 describes the implementation of the algorithm in SWI Prolog and the projection at work, and Section 6 concludes.

Motivating scenario. In order to better understand the impact of distributed monitoring of complex and open systems, let us consider the following scenario: a humanitarian convoy in charge of food transportation is traversing a potentially hostile country. In order to ensure the convoy safety, a set of autonomous unmanned aerial vehicles (UAV) is deployed. The goals assigned to the UAVs are as diverse as: 1. maintaining the convoy within sight of a distant control center thanks to an embedded camera and data transmission; 2. transmitting images of the situation ahead of the convoy (to the convoy itself and to the control center); 3. ensuring data transmission from the convoy to the external world and conversely; 4. detecting potential hazards and informing the convoy and the control center; 5. localizing suspicious vehicles; 6. identifying a designated mobile entity, etc.

Several UAVs are required to achieve some of these goals since they require being at different locations at the same time (goals 1, 2). On the contrary, some goals can be assigned to the same UAV, providing the UAV traveling from one specific location to another one (goals 4, 5, 6). Moreover, some goals can be shared between UAVs (goal 3). When some UAV becomes unavailable, its goals must be allocated to another one or a new UAV must take-off depending on the

[1] `http://jason.sourceforge.net/wp/`
[2] `http://jade.tilab.com/`

resources availability. It is the case when communication failures occur, which might be temporary or permanent. It is also the case of instrument failure on-board UAVs, of meteorological events, etc. Due to situation-related hazards, the convoy might (autonomously or by a decision coming from the control center) decide to change its route. This change has to be taken into account by all the UAVs, which implies at the same time a re-planning of UAVs trajectories but also re-planning of the tasks they have been allocated to since their feasibility is not anymore ensured (fuel resources, communication network, etc.). It is of a major importance that the protocols implemented in the system are monitored for two reasons: 1. possible errors in protocols might generate confusion among agents and generate bad decisions whose consequences might be dramatic; 2. malevolent actors might try to penetrate the system since humanitarian operations almost often occur in a tense political context.

Unfortunately, a centralized monitoring is difficult to carry out in such a system since it forces every agent to communicate with a unique control agent, which is not always possible due to the physical dispersion of the agents. For example, a low altitude UAV can only communicate with a distant control center in gaining altitude, which is incompatible with a permanent monitoring of its communications since most of the UAV mission takes place close to the ground. Hence, in an application such as the humanitarian convoy the distribution of protocol monitoring and the ability of any agent to monitor part of the protocol, if needed, is a problem that must be addressed. It is not a surprise since the functions of the application themselves have to be implemented as autonomous goal-directed agents to be able to tackle the complexity inherent to this kind of systems. Adding a centralized monitoring is then hopeless.

2 State of the Art

In this section we review the literature on runtime monitoring of interaction protocols, on the distribution of monitoring among subsets of components with a specific attention to how decentralized monitoring can ensure global protocol compliance, and on projections that move from global types to global types in order to lighten them.

Runtime monitoring of interaction protocols. Many frameworks and formalisms for monitoring the runtime execution of a distributed system have been proposed in the last years.

One of the most recent and relevant work in this area is SPY (Session Python) [24], a tool chain for runtime verification of distributed Python programs against protocol specifications expressed in Scribble[3]. Given a Scribble specification of a global protocol, the tool chain validates consistency properties, such as race-free branch paths, and generates Scribble (i.e. syntactic) local protocol specifications for each participant (role) defined in the protocol. At runtime, an independent monitor (internal or external) is assigned to each Python endpoint and verifies

[3] `http://www.scribble.org`

the local trace of communication actions executed during the session. That work shares motivations similar to ours. The main differences lie in the expressive power of the two languages, which is higher for our formalism of constrained global types due to the constrained shuffle operator which is missing in Scribble, and in the availability of tools for statically verifying properties of Scribble specifications, which are not available for constrained global types.

Many other approaches for runtime monitoring of distributed systems and MASs exist like those mentioned in the sequel, but with no emphasis on the projection from global to local monitors. This represents the main difference between those proposals and ours.

In [17], aspect-oriented development techniques are used to enhance existing code of runtime monitors, checking the interaction behavior of applications against their specifications. Message Sequence Charts (MSCs) are exploited to specify the interaction behavior of distributed systems and as a basis for automatic runtime monitor generation. An explanation of the monitor generation procedure and tool set is presented using a case study from the embedded automotive systems domain. Addressing the need for formal specification and runtime verification of system-level requirements of distributed reactive systems, [14] presents a formalism for specifying global system behaviors in terms of MSCs assertions, with a technique for the evaluation of the likelihood of success of a distributed protocol under non-trivial communication conditions via discrete event simulation and runtime execution monitoring.

Moving to the MAS field, a great attention has been recently devoted to monitoring norms and commitments: formalizing the entities participating to a protocol and the rules regulating their interaction is in fact an inherent aspect of normative systems. In [23] a generic architecture for observing agent behaviors and recognizing those which comply to or violate the predefined norms is described. The architecture deploys monitors that receive inputs from observers and process these inputs together with transition network representations of individual norms. In this way, monitors determine the fulfillment or violation status of norms. As far as commitments are concerned, one of the first contributions were Commitment Machines [29], a formalism modeling communication protocols supplying a content to protocol states and actions in terms of the social commitments of the participants. The content can be reasoned about by the agents, thereby enabling flexible execution of the given protocol. In [27] Distributed Commitment Machines are defined and the properties of Commitment Machines, both distributed and centralized, are explored. A recent work on relationship between agents and commitment-based protocols is [12], where the authors specify agents in terms of goal models and protocols in terms of commitments among agents. The semantic relationship between agents and protocols is formalized exploiting the relationship between goals and commitments. Given an agent specification and a protocol, it is possible to verify whether the protocol allows the achievement of particular agent goals, and whether the agent's specification supports the satisfaction of particular commitments. In [4] commitments are exploited again in normative MASs: the authors focus on JADE and show

that it is possible to account for interactions by exploiting commitment-based protocols, by modifying the Jade Methodology so as to include the new features in a seamless way, and by relying on the notion of artifact.

In [15] a framework for automatic processing of interactions generated using FIPA-ACL[4] is presented. This framework includes three elements: i) an agent interaction architecture to systematize interaction processing tasks, ii) interaction models to build re-usable validated code used to check different phases of interaction processing associated with message semantics, and iii) components and control structures implementing interaction architecture for a particular agent platform. The paper describes the implementation details of the proposed approach developed within the CAPNET agent platform.

Finally, [22] describes an architecture for verifying properties of a multiagent system during its execution. Considering that a correct system is a system verifying the properties specified by the designer, the authors focus on the "property" notion. The architecture, a MAS itself, is based on a set of agents whose goals are to check at runtime the whole system's properties.

Compliance of distributed and centralized monitoring. The problem of distributed monitoring has been faced by many researchers in MASs, web services, sensor networks and other distributed systems, but often the proposed solutions either directly describe a distributed protocol without any central point of control, or dynamically create groups of entities (agents, services, components) for monitoring different areas with no central representation of the global protocol, making these approaches and their theoretical foundations not comparable with ours.

Also, some proposals are similar to ours, but no formal justification of the projection and its coherence with the global protocol is provided. For example, the idea of "splitting" a global protocol into subprotocols has been proposed thirty years ago in the area of network communication protocols [18] and more recently in the one of Web Services choreography [25], but without a theoretical basis.

The formalization and analysis of the relation between a global description of a distributed system and a more machine-oriented description of a set of components that implements it is a problem that has been studied in several contexts and by different communities, as widely discussed for example in the related work section of [10]. Projecting a global protocol into a stub of an executable piece of code, or - on the other way round - verifying at design time that an executable piece of code respects the global protocol specification are problems different from what we face: we do not need to know the implementation of the agents in order to perform a runtime verification of their observable behavior with respect to the global protocol, and we project global protocols involving many agents into sub-protocols involving less, and not global protocols into "implementations".

Although different from ours, contributions dealing with global types projected onto session types and their declination as choreographies projected onto contracts, can be a source of inspiration for identifying the syntactic and semantic conditions which make the projection of a constrained global type feasible.

[4] `http://www.fipa.org/specs/fipa00061/SC00061G.html`

Global types [9,10,16] are behavioral types, whose aim is the specification and verification of multiparty interactions between distributed components. As suggested by the term "global", they describe the overall communication behavior of a distributed system, whereas session types specify the behavior of the single components of a system.

In [10] Castagna et al. tackle the problem of projecting global to session types; in particular, projection is well-defined only if well-formedness conditions are satisfied by global types. Such non trivial conditions are expressed in terms of the semantics of global types, which corresponds to sets of traces. The defined projection algorithm is not complete, since it is not defined for all global types that satisfy the semantic conditions for projectability.

Global types can be seen as web service choreographies[5] describing the interaction of some distributed processes connected through a private multi-party session. Therefore, there is a close relationship between the work of Castagna et al., and those by Zavattaro et al. [6,19] which concern the projection of choreographies into the contracts of their participants. The projection procedure is basically an homomorphism from choreographies to the behavior of their participants. While [7] gives no conditions to establish which choreographies produce correct projections, [19] defines three connectedness conditions that guarantee correctness of the projection for various (synchronous and asynchronous) semantics, solely stated on the syntax of the choreography.

The problem of analyzing choreographies and characterizing their properties has been addressed also by the MAS community. In particular, Baldoni et al. [5] propose a notion of interoperable choreography which basically coincides with Castagna et al.'s notion of liveness: the interaction between the parties must preserve the ability to reach a state in which every party has successfully completed its task. Also the notion of conformance between parties defined by Baldoni et al. may be a basis for proposing methods and algorithms for devising whether a set of projected protocols expressed in our formalism for constrained global types can be used to verify the same properties as the global one.

Lightening global types. As seen in the previous paragraph, projection of global types (resp. choreographies) usually moves from global to session types (resp. from choreographies to contracts). A very recent proposal by T-C. Chen [11] shares with ours the purpose of moving from global types to global types, in order to "lighten" the original global type.

The motivation for Chen's work is that some interactions in global types take place just for the purpose of informing receivers that some message will never arrive or the session is terminated. By decomposing a big global type into several simpler global types, one can avoid such kind of redundant interactions. Chen proposes a framework for easily decomposing global types into light global types, preserving the interaction sequences of the original ones but for redundant interactions.

[5] `http://www.w3.org/TR/ws-cdl-10/`

Although the rationale for our "lightening" function is to remove interactions not involving some agents rather than removing redundant interactions as in Chen's work, her proposal is the only one, to the best of our knowledge, where projection moves from global types to global types. While Chen demonstrates the correctness of her lightening function, she did not implement it yet. Conversely, our projection function is implemented and usable by both JADE and Jason agents, although we did not formally demonstrate its properties yet.

3 Backgroud

This section briefly recaps on constrained global types, omitting their extension with attributes [20] because the projection algorithm discussed in Section 4 is currently defined on "plain" constrained global types only.

Constrained global types (also named "types" in the sequel, when no ambiguity arises) are defined starting from the following entities:

Interactions[6]. An interaction a is a communicative event taking place between two agents. For example, `msg(right_robot, right_monitor, tell, put_sock)` is an interaction involving the sender `right_robot` and the receiver `right_monitor`, with performative `tell` and content `put_sock`.

Interaction types. Interaction types model the message pattern expected at a certain point of the conversation. An interaction type α is a predicate on interactions. For example, `msg(right_robot, right_monitor, tell, put_sock)` $\in$ `put_right_sock` means that interaction `msg(right_robot, right_monitor, tell, put_sock)` has type `put_right_sock`.

Producers and consumers. In order to model constraints across different branches of a constrained fork, we introduce two different kinds of interaction types, called *producers* and *consumers*, respectively. Each occurrence of a producer interaction type must correspond to the occurrence of a new interaction; in contrast, consumer interaction types correspond to the same interaction specified by a certain producer interaction type. The purpose of consumer interaction types is to impose constraints on interaction traces, *without introducing new events*. A consumer is an interaction type, whereas a producer is an interaction type α equipped with a natural superscript n specifying the exact number of consumer interactions which are expected to coincide with it.

Constrained global types. A constrained global type τ represents a set of possibly infinite traces of interactions, and is a possibly cyclic term defined on top of the following type constructors:

- λ (empty trace), representing the singleton set $\{\epsilon\}$ containing the empty trace ϵ.
- $\alpha^n{:}\tau$ (*seq-prod*), representing the set of all traces whose first element is an interaction a matching type α ($a \in \alpha$), and the remaining part is a trace

[6] "Interactions" were named "sending actions" in our previous work. We changed terminology to be consistent with the one used in the choreography community.

in the set represented by τ. The superscript[7] n specifies the number n of corresponding consumers that coincide with the same interaction type α; hence, n is the least required number of times $a \in \alpha$ has to be "consumed" to allow a transition labeled by a.
- $\alpha{:}\tau$ (*seq-cons*), representing a consumer of interaction a matching type α ($a \in \alpha$).
- $\tau_1 + \tau_2$ (*choice*), representing the union of the traces of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*fork*), representing the set obtained by shuffling the traces in τ_1 with the traces in τ_2.
- $\tau_1 \cdot \tau_2$ (*concat*), representing the set of traces obtained by concatenating the traces of τ_1 with those of τ_2.

Since constrained global types are interpreted coinductively [1], it is possible to specify protocols that are not allowed to terminate like for example the `PingPong` protocol defined by the equation

```
PingPong = (ping,0):(pong,0):PingPong
```

where `PingPong` is a logical variable which is unified with a recursive (or cyclic, or infinite) term consisting of the producer interaction type `ping`, followed by the producer interaction type `pong` (both requiring 0 consumers), followed by the term itself. The coinductive interpretation (that is, the greatest fixed point of the function F corresponding to the recursive definition of `PingPong`) is the singleton containing the only valid and infinite interaction trace `ping pong ping pong ping pong ...`. The inductive interpretation (that is, the least fixed point of F) of `PingPong` is the empty set, since there is no base for the induction; hence, coinduction [26] is required for correctly dealing with infinite traces.

The valid traces for the type

```
PingPong = ((ping,0):(pong,0):PingPong) + lambda
```

instead, are {ϵ, `ping pong`, `ping pong ping pong`, ...}, namely all the traces consisting of an arbitrary number (even none or infinite) of `ping pong`.

Let us consider the following simple example where there are two robots (right and left), two monitors (right and left) associated with each robot, and a plan monitor which supervises them (Figure 1). The goal of the MAS is to help mothers in speeding up dressing their kids by putting their shoes on: robots must put a sock and a shoe on the right (resp. left) foot of the kid they help. As robots are autonomous, they could perform the two actions in the wrong order, making the life of the mothers even crazier... 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. Each robot can start by putting the sock, which is the correct action to do, or by putting the shoe, which requires a recovery by the (right or left, resp.) robot monitor.

[7] In the examples throughout the paper we use the concrete syntax of Prolog where producer interaction types are represented by pairs (α,n).

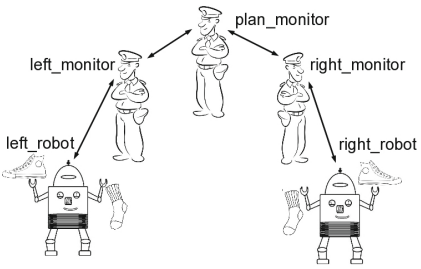

Fig. 1. The "socks and shoes" MAS

As we will see, the left and right monitors play two different roles: they interact with robots to detect wrong actions and recover them, and they also verify part of the protocol, notifying the user of protocol violations. In this MAS, *monitors are part of the protocol itself*. In the MASs described in our previous papers, monitors performed a runtime verification of all the other agents but themselves, and their sole goal was to detect and signal violations. Extending monitors with other capabilities (or, taking another perspective, extending "normal" agents with the capability to monitor part of the protocol) does not represent an extension of the language or framework. The possibility of having agents that can monitor, can be monitored, and can perform whatever other action, was already there, but we did not exploit it before.

The interactions involved in the socks and shoes protocol and their types are as follows:

```
msg(right_robot, right_monitor, tell, put_sock) ∈ put_right_sock
msg(right_robot, right_monitor, tell, put_shoe) ∈ put_right_shoe
msg(right_robot, right_monitor, tell, removed_shoe) ∈ rem_right_shoe
msg(right_monitor, right_robot, tell, obl_remove_shoe) ∈ obl_rem_right_shoe
msg(right_monitor, plan_monitor, tell, ok) ∈ ok_right
msg(left_robot, left_monitor, tell, put_sock) ∈ put_left_sock
msg(left_robot, left_monitor, tell, put_shoe) ∈ put_left_shoe
msg(left_robot, left_monitor, tell, removed_shoe) ∈ rem_left_shoe
msg(left_monitor, left_robot, tell, obl_remove_shoe) ∈ obl_rem_left_shoe
msg(left_monitor, plan_monitor, tell, ok) ∈ ok_left
```

The protocol can be specified by the following types, where `SOCKS` corresponds to the whole protocol.

```
RIGHT = ((put_right_sock,0):(put_right_shoe,0):(ok_right,0):lambda) +
((put_right_shoe,0):(obl_rem_right_shoe,0):(rem_right_shoe,0):RIGHT),
LEFT = ((put_left_sock,0):(put_left_shoe,0):(ok_left,0):lambda) +
((put_left_shoe,0):(obl_rem_left_shoe,0):(rem_left_shoe,0):LEFT),
SOCKS = (RIGHT | LEFT)
```

The type `SOCKS` specifies the shuffle (symbol "|") of two sets of traces of interactions, corresponding to `RIGHT` and `LEFT`, respectively. The shuffle expresses the

fact that interactions in `RIGHT` are independent (no causality) from interactions in `LEFT`, and hence traces can be mixed in any order.

Types `RIGHT` and `LEFT` are defined recursively, that is, they correspond to cyclic terms. `RIGHT` consists of a choice (symbol "+") between the finite trace (the constructor for trace is ":") of interaction types `(put_right_sock,0)`, `(put_right_shoe,0)`, `(ok_right,0)` corresponding to the correct actions of the right robot, and the trace of interaction types `(put_right_shoe,0)`, `(obl_rem_right_shoe,0)`, `(rem_right_shoe,0)` corresponding to the wrong initial action of the robot, followed by an attempt to perform the `RIGHT` branch again. Basically, either the right robot tells the right monitor that it put the sock on first, and then it can go on by putting the shoe, or it tells that it started its execution by putting the shoe on. In this case, the right monitor forces the robot to remove the shoe, the robot acknowledges that it removed the shoe, and then starts again. The `LEFT` branch is the same as the `RIGHT` one, but involves the left robot and the left node monitor.

An example where sets of traces could be expressed with a fork, but are not completely independent, is given by the Alternating Bit Protocol ABP. We

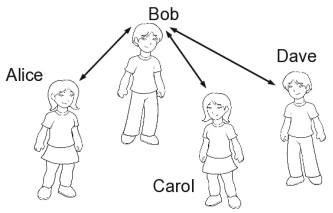

Fig. 2. The ABP3 MAS

consider the instance of ABP where six different sending actions may occur (Figure 2): Bob sends `msg1` to Alice (interaction type `m1`), Alice sends `ack1` to Bob (sending action type `a1`), Bob sends `msg2` to Carol (interaction type `m2`), Carol sends `ack2` to Bob (sending action type `a2`), Bob sends `msg3` to Dave (interaction type `m3`), Dave sends `ack3` to Bob (interaction type `a3`). The ABP is an infinite iteration, where the following constraints have to be satisfied for all occurrences of the sending actions:

– The n-th occurrence of an interaction of type `m1` must precede the n-th occurrence of an interaction of type `m2` which in turn must precede the n-th occurrence of an interaction of type `m3`.

– 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 .

The ABP cannot be specified with forks of independent interactions, hence a possible solution requires to take all the combinations of interactions into account in an explicit way. However with this solution the size of the type grows exponentially with the number of the different interaction types involved in the protocol.

With producer and consumer interaction types it is possible to express the shuffle of non independent interactions which have to verify certain constraints. In this way the ABP can be specified in a very compact and readable way. The whole protocol is specified by the following constrained global type ABP3:

```
M1M2M3=m1:m2:m3:M1M2M3,
M1A1=(m1,1):(a1,0):M1A1,
M2A2=(m2,1):(a2,0):M2A2,
M3A3=(m3,1):(a3,0):M3A3,
ABP3=((M1M2M3|M1A1)|(M2A2|M3A3))
```

Fork is associative and the way we put brackets in ABP3 does not matter: ((M1M2M3|M1A1)|(M2A2|M3A3)) has the same meaning as (M1M2M3|(M1A1|M2A2)|M3A3), and as any other association.

4 Projection Algorithm

In the "socks and shoes" example the monitors, besides checking that the robots accomplish their goal, verify also the compliance of the system to the specification of the protocol, given by the type SOCKS. If we assume that the right robot and the right monitor reside on the same node, then it is reasonable that the right monitor verifies only the interactions which are local to its node; to do that, we must project the type SOCKS onto the agents of the node, that is, the right robot and the right monitor.

What we would like to obtain is the type

```
RIGHT_P = ((put_right_sock,0):(put_right_shoe,0):(ok_right,0):lambda) +
((put_right_shoe,0):(obl_rem_right_shoe,0):(rem_right_shoe,0):RIGHT_P),
SOCKS_P = (RIGHT_P|lambda)
```

which only contains interactions where the right robot and the right monitor are involved, either as sender or as receiver.

We can project any protocol onto any set of agents (although it is not necessarily meaningful or useful). For example, projecting the ABP3 on Dave should result into

```
ABP3_P_compact = (m3,0):(a3,0):ABP3_P_compact
```

which just states that Dave must ensure to respect the order between messages of type m3 and acknowledges of type a3 between him and Bob. That projected type can be represented in an equivalent way, even if less compact, as

```
M1M2M3_P = m3:M1M2M3_P,
M3A3_P = (m3,1):(a3,0):M3A3_P,
ABP3_P =((M1M2M3_P|lambda)|(lambda|M3A3_P))
```

Projecting the ABP3 on Bob, instead, should result into the ABP3 itself as Bob is involved in all communications and hence no interaction will be removed from the projection.

Since Dave cannot be aware of the order among messages from other agents to Bob, he can only monitor a part of the protocol. Therefore, distributing the ABP among Alice, Carol and Dave would result in a partial verification of the protocol not able to detect all possible errors; indeed, Bob is necessary for checking the constraints involving `m1`, `m2`, `m3`, and, hence, is the only agent that can monitor the protocol.

In order to allow agents to verify only a sub-protocol of the global interaction protocol, we designed a projection algorithm that takes a constrained global type and a set of agents Ags as input, and returns a constrained global type which contains only interactions involving agents in Ags. The intuition besides the algorithm is that interactions that do not involve agents in Ags are removed from the projected constrained global type. Given the finite set AGS of all the agents that could play a role in the MAS and an interaction type α, $senders(\alpha)$ is the set of all the agents in AGS that could play the role of sender in actual interactions having type α and $receivers(\alpha)$ 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, $involves(\alpha, Ags)$, iff $(senders(\alpha) \subseteq Ags) \vee (receivers(\alpha) \subseteq Ags)$.

Projection can be described as a function $\Pi : \mathcal{CT} \times \mathcal{P}(AGS) \rightarrow \mathcal{CT}$ where $\mathcal{CT}$ is the set of constrained global types. Π is driven by the syntax of the type to project[8]; since Π is defined on cyclic terms, the simplest way to define it would be by coinduction as follows:

(i) $\Pi(\lambda, Ags) = \lambda$
(ii) $\Pi(\alpha : \tau, Ags) = \alpha : \Pi(\tau, Ags)$ if $involves(\alpha, Ags)$
(iii) $\Pi(\alpha : \tau, Ags) = \Pi(\tau, Ags)$ if $\neg involves(\alpha, Ags)$
(iv) $\Pi(\tau' \ op \ \tau'', Ags) = \Pi(\tau', Ags) \ op \ \Pi(\tau'', Ags)$, where $op \in \{+, |, \cdot\}$.

However, this definition is not fully correct: it works properly on non cyclic terms (example 1) and on some cyclic terms (example 2), but does not behave correctly with other kinds of cyclic terms as shown in examples 3 and 4.

Example 1 (non cyclic terms). Let us consider a simple non cyclic term T defined by $T = a : b : \lambda$. We want to project T on Ags. Suppose that $involves(a, Ags)$ holds, whereas $involves(b, Ags)$ does not (this assumption will hold for the following examples too), meaning that interaction type a must be kept in the projection and b must be removed. From (ii) we get $\Pi(a : b : \lambda, Ags) = a : \Pi(b : \lambda, Ags)$ (a is kept in the projection), from (iii) we have $\Pi(b : \lambda, Ags) = \Pi(\lambda)$ (b is discarded from the projection), and finally, from (i) we know that $\Pi(\lambda) = \lambda$, therefore $\Pi(T, Ags) = a : \lambda$.

Example 2 (cyclic terms without problems). Let us now consider the cyclic term T s.t. $T = a : T'$ and $T' = b : T$.

[8] In the sequel of this section we will use "type" and "term" interchangeably, as a constrained global type (or just type) is represented by a term.

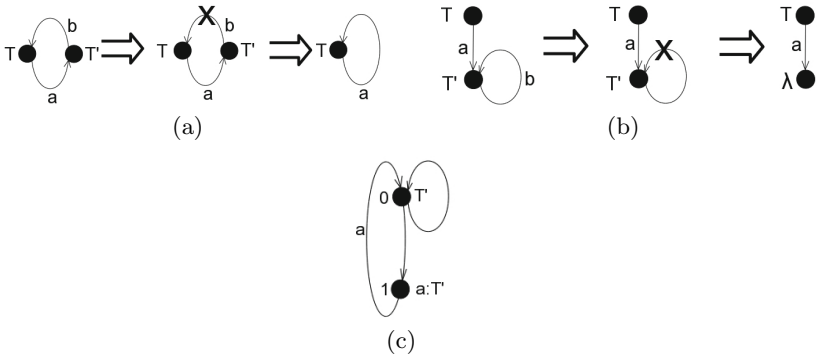

Fig. 3. Correct projection of a cyclic term

Again, the projection is driven by the syntax of T; by applying the definition of Π we have given before, we have $\Pi(a : T', Ags) = a : \Pi(T', Ags) = a : \Pi(b : T, Ags) = a : \Pi(T) = a : \Pi(a : T', Ags)$; while in the previous, non recursive example we could conclude by applying the definition $\Pi(\lambda, Ags) = \lambda$ corresponding to the λ type, in this case we do not have any basis. However, by coinduction we can conclude that $\Pi(a : T', Ags)$ has to return the unique cyclic term T'' s.t. $T'' = a : T''$ (see Figure 3(a)), which corresponds to the correct projection.

Example 3 (cyclic terms with problems - non uniqueness). The definition of Π needs to be refined because it does not always specify a unique result; to see that, let us consider the cyclic term T s.t. $T = a : T'$ and $T' = b : T'$ with the same definition of *involves* as before. Now from the definitions above we get $\Pi(a : T', Ags) = a : \Pi(T', Ags)$, $\Pi(T', Ags) = \Pi(b : T', Ags) = \Pi(T', Ags)$; since $\Pi(T', Ags) = \Pi(T', Ags)$ is an identity, Π is allowed to return any type when applied to T'[9], while the expected correct type should be λ, so that $\Pi(a : T', Ags) = a : \lambda$ (see Figure 3(b)). This example demonstrates that the definition of Π as given before must be reconsidered for coping with cases like this one correctly (see the paragraph "Projection function refined" below).

Example 4 (cyclic term with problems - non contractiveness). Finally, let us consider the cyclic term T s.t. $T = (a : T) + (b : T)$; by (iv) $\Pi(T, Ags) = \Pi(a : T, Ags) + \Pi(b : T, Ags)$, by (ii) $\Pi(a : T, Ags) = a : \Pi(T, Ags)$, and by (iii) $\Pi(b : T, Ags) = \Pi(T, Ags)$, therefore by coinduction the returned type is T' s.t. $T' = (a : T') + T'$; although in this case there exists a unique type that can be returned by Π, such a type is not *contractive*. A type is contractive if all possible cycles in it contain an occurrence of the sequence constructor ":"; Figure 3(c) shows that type T' s.t. $T' = (a : T') + T'$ is not contractive, since the rhs cycle contains only the "+" operator.

Contractive types ensure that runtime verification always terminates and we want that contractive constrained global types like T s.t. $T = (a : T) + (b : T)$ are always projected into contractive constrained global types. The refinement of Π discussed below copes with this requirement as well.

[9] In the same way as the equation $X = X$ is satisfied for any value associated with X.

Projection function refined. To guarantee that the projection function always returns a contractive type and that the correct coinductive definition is implemented, we need to keep track of all types visited by Π along a path[10]; 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 (and we can detect this situation because we keep track of all the already visited types, together with their depth and projection), 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 λ is returned.

Let us consider again the type $T = (a : T) + (b : T)$ from example 4; when computing its projection, the depth of T is 0, and initially we set the value of *DeepestSeq* to -1. When visiting the lhs path starting from the "+" operator, the type $a : T$ is visited at depth 1, and *DeepestSeq* is set to 1, since the root of $a : T$ is the sequence constructor. Then T is revisited, and since its depth 0 is less than *DeepestSeq*, the projection of the lhs is $T' = a : T'$. When visiting the rhs path starting from the "+" operator, *DeepestSeq* contains again the value -1, and the type $b : T$ is visited at depth 1, but because $involves(b, Ags)$ does not hold, b is discarded with the corresponding sequence constructor, hence *DeepestSeq* is not updated. Then T is revisited, and since its depth 0 is not less than *DeepestSeq*, the projection of the rhs is λ. The next section provides a detailed description of the implementation of the correct projection algorithm.

5 Implementation and Use

In this section, we show Π's implementation and we frame it into our framework for distributed runtime verification of MASs. The framework, depicted in Figure 4, consists of four layers: **(1)** a formalism for describing agent interaction protocols (AIPs) based on constrained global types, along with an algorithm to validate at design time that the described protocol models the expected traces of interaction; **(2)** the projection algorithm, along with a generate and test algorithm for validating at design time that the projection on a given agents' subset can be safely used for dynamic verification; **(3)** a mechanism for verifying at runtime that interactions are compliant with the AIP; and **(4)** a mechanism for intercepting at runtime actual messages involving the agents under monitoring, be them JADE or Jason ones, in a way as transparent as possible.

Whereas the design time validation algorithms supporting layers 1 and 2 can only generate and test traces of finite length, the runtime verification of layer 3 could in principle go on forever, if the protocol is an infinite one: the runtime verification mechanism checks the compliance of each actual interaction taking

[10] By "path" we mean the path in the tree associated with the type; for example, if the type is T s.t. $T = (a : T) + (b : T)$, Π will first visit the path associated with $(a : T)$ and then that associated with $(b : T)$.

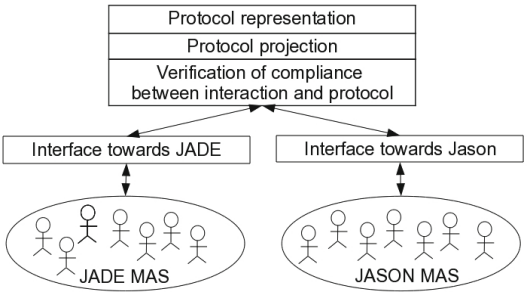

Fig. 4. Our modular framework for distributed runtime verification of MASs

place in the MAS w.r.t. the constrained global time and stops only when a violation is detected.

The choice of JADE and Jason as the two frameworks that we are able to monitor is due to their widespread adoption in the agent community.

Implementation. The projection algorithm has been implemented in SWI Prolog, `http://www.swi-prolog.org/`, which manages infinite terms in an efficient way. Since we need to record the association between any type and its projection in order to correctly detect and maage cycles, we exploited the SWI Prolog library `assoc` for association lists, `http://www.swi-prolog.org/pldoc/man?section=assoc`. The three predicates of the library `assoc` that we use for our implementation are

- `empty_assoc`(*-Assoc*): *Assoc* is unified with an empty association list.
- `get_assoc`(*+Key, +Assoc, ?Value)*: *Value* is the value associated with *Key* in the association list *Assoc*.
- `put_assoc`(*+Key, +Assoc, +Value, ?NewAssoc*): *NewAssoc* is an association list identical to *Assoc* except that *Key* is associated with *Value*. This can be used to insert and change associations.

The projection is implemented by a predicate `project(T, ProjAgs, ProjT)` where `T` is the constrained global type to be projected, `ProjT` is the result, and `ProjAgs` is the set of agents onto which the projection is performed. The algorithm exploits the predicate `involves(IntType, ProjAgs)` succeeding if `IntType` may involve one agent, as a sender or a receiver, in `ProjAgs`.

Currently `involves` looks for actual interactions `ActInt` whose type is `IntType` and assumes that senders and receivers in `ActInt` are ground terms, but it could be extended to take agents' roles into account or in other more complex ways. It uses the "or" Prolog operator `;` and the `member` predicate offered by the library `lists`. It exploits the predicate `has_type(ActInt, IntType)` implementing the definition of the type `IntType` of an actual interaction `ActInt`.

```
involves(IntType, List) :-
has_type(msg(Sender, Receiver, _, _), IntType),
(member(Sender, List);member(Receiver, List)).
```

For the implementation of `project/3` we use an auxiliary predicate `project/6` with the following three additional arguments:

- an initially empty association `A` to keep track of cycles;
- the current depth of the constrained global type under projection, initially set to `0`;
- the depth of the deepest sequence operator belonging to the projected type, initially set to `-1`.

```
project(T, ProjAgs, ProjT) :-
empty_assoc(A), project(A, 0, -1, T, ProjAgs, ProjT).
```

The predicate is defined by cases.

1. `lambda` is projected into `lambda`.

```
project(_Assoc, _Depth, _DeepestSeq, lambda, _ProjAgs, lambda):- !.
```

2. If `Type` has been already met while projecting the global type (`get_assoc (Type, Assoc, (AssocProjType,LoopDepth))` succeeds), then its projection `ProjT` is `AssocProjType` if `LoopDepth =< DeepestSeq` and is `lambda` otherwise. The "if-then-else" construct is implemented in Prolog as `Condition -> ThenBranch ; ElseBranch`.

```
project(Assoc, _Depth, DeepestSeq, Type, _ProjAgs, ProjT) :-
get_assoc(Type,Assoc,(AssocProjType,LoopDepth)),!,
(LoopDepth =< DeepestSeq -> ProjT=AssocProjType; ProjT=lambda).
```

3. `T = (IntType:T1)`. `IntType` is a consumer since it has no integer number associated with it. `ProjT` is recorded in the association `A` along with the current depth `Depth` (`put_assoc((IntType:T1),Assoc,(ProjT,Depth), NewAssoc)`). If `IntType` involves `ProjAgs`, `ProjT=(IntType:ProjT1)` where `ProjT1` is obtained by projecting `T1` onto `ProjAgs`, with association `NewAssoc`, depth of the type under projection increased by one, and depth of the deepest sequence operator equal to `Depth`. If `IntType` does not involve `ProjAgs`, then the projection on `T` is the same as `T1` with association `NewAssoc`, depth of the type under projection equal to `Depth`, and depth of the deepest sequence operator equal to `DeepestSeq`.

```
project(Assoc, Depth, DeepestSeq, (IntType:T1), ProjAgs, ProjT) :- !,
put_assoc((IntType:T1),Assoc,(ProjT,Depth),NewAssoc),
(involves(AMsg, ProjAgs) ->
IncDepth is Depth+1,
project(NewAssoc,IncDepth,Depth,T1,ProjAgs,ProjT1),
ProjT=(IntType:ProjT1);
project(NewAssoc,Depth,DeepestSeq,T1,ProjAgs,ProjT)).
```

4. `T = ((IntType,N):T1)`. `(IntType,N)` is a producer since it has an integer number `N` associated with it. The clause for projection is identical to the previous case, except for the atom `ProjT=(IntType:ProjT1)` in the first branch of the condition which becomes `ProjT=((IntType,N):ProjT1)`.

5. `T = T1 op T2`, where `op` $\in$ {`+`, `|`, `*`}: the association between `T1 op T2` and the projected type `ProjT` is recorded in the association `Assoc` along with the current depth `Depth`, `T1` and `T2` are projected into `ProjT1` and `ProjT2` respectively, with association equal to `NewAssoc`, depth of the type under projection increased by one and depth of the deepest sequence operator equal to `DeepestSeq`. The result of the projection is `ProjT=(ProjT1 op ProjT2)`. For example, if `op` is `+`, the Prolog clause is:

```
project(Assoc, Depth, DeepestSeq, (T1+T2), ProjAgs, ProjT) :- !,
put_assoc((T1+T2),Assoc,(ProjT,Depth),NewAssoc),
IncDepth is Depth+1,
project(NewAssoc, IncDepth, DeepestSeq, T1, ProjAgs, ProjT1),
project(NewAssoc, IncDepth, DeepestSeq, T2, ProjAgs, ProjT2),
ProjT=(ProjT1+ProjT2).
```

Types `SOCKS_P` and `AP3_P` shown at the beginning of Section 4 have been obtained by applying the projection algorithm to types `SOCKS` and `ABP3` respectively. The reason why they are not as compact as possible, which is mainly evident in `AP3_P`, is that the projection algorithm does not implement a further simplification step and hence some types which have been projected into `lambda` could have been safely removed.

The result of the projection may be a type equivalent to `lambda`. For example, if we project `ABP` to the set {`eric`}, no interaction involves it and the result is `(lambda|lambda)|lambda|lambda`. Optimizing the algorithm to perform this simplification step is a forthcoming improvement, easy to face in Prolog. On the other hand, we have already observed that the projection may be the same as the projected type. This happens for example if we project `ABP` to the set {`bob`}, which interacts with all the agents in the MAS.

Design time validation that centralized protocol behaves as expected. In SWI Prolog we have implemented a mechanism for generating all the different traces (sequences of interactions) with length N, where N can be set by the user, that respect a given protocol. This mechanism is necessary during the design of the protocol and allows the protocol designer to make an empirical assessment of the conversations that will be recognized as valid ones during the runtime verification. We used this mechanism for validating the "centralized" protocols.

For example, Table 1 (left) shows one of the 16380 different traces with length 12 of the SOCKS protocol (for sake of presentation, we abbreviate `right_robot` in `right_r`, `right_monitor` in `right_m`, `left_robot` in `left_r`, `left_monitor` in `left_m`, `msg` in `m`, and we drop the `tell` performative from interactions). The trace corresponds to an execution where the protocol reached a final state and no other interactions could be accepted after the last one. In the output produced by the SWI Prolog algorithm, this information is given by means of an asterisk after the last interaction. Traces that are prefixes of longer (maybe infinite) ones have no asterisk at their end. Table 1 (right) shows one of the 30713 different traces with length 16 of the ABP3 protocol. Since the ABP3 is an infinite protocol, all its traces are prefixes of infinite ones.

Table 1. Traces of the SOCKS and ABP "centralized" protocols

SOCKS protocol	ABP protocol
m(right_r, right_m, put_sock)	msg(bob, alice, tell, m1)
m(left_r, left_m, put_shoe)	msg(bob, carol, tell, m2)
m(left_m, left_r, oblige_remove_shoe)	msg(carol, bob, tell, a2)
m(left_robot, left_m, removed_shoe)	msg(alice, bob, tell, a1)
m(right_r, right_m, put_shoe)	msg(bob, dave, tell, m3)
m(right_m, plan_monitor, ok)	msg(dave, bob, tell, a3)
m(left_robot, left_m, put_shoe)	msg(bob, alice, tell, m1)
m(left_m, left_r, oblige_remove_shoe)	msg(bob, carol, tell, m2)
m(left_r, left_m, removed_shoe)	msg(alice, bob, tell, a1)
m(left_r, left_m, put_sock)	msg(bob, dave, tell, m3)
m(left_r, left_m, put_shoe)	msg(bob, alice, tell, m1)
m(left_m, plan_monitor, ok)	msg(carol, bob, tell, a2)
*	msg(dave, bob, tell, a3)
	msg(bob, carol, tell, m2)
	msg(alice, bob, tell, a1)
	msg(carol, bob, tell, a2)

By generating traces of different length and inspecting some of them, the protocol designer can get a clear picture of whether the protocol he/she designed behaves in the expected way. Of course this manual inspection gives no guarantees of the correctness of the protocol specification, but in our experience it was enough to early detect flaws.

Design time validation that the projected protocol makes sense. This step was devised for giving hints on whether the decentralized monitoring can ensure global protocol compliance. In fact, although all well-formed deterministic and contractive constrained global types can be projected, not all possible partitions of a subset of all agents of the system to be verified allows a full distributed monitoring of the protocol's properties.

For example, in the case of the SOCKS protocol, deciding which were the subsets of agents onto which projecting the global protocol in order to distribute the monitoring activity was easy: interactions induce a graph connecting pairs of agents that interact at some point, and in this case the graph is a tree as shown in Figure 1. By projecting onto {`left_monitor`} and allowing `left_monitor` to monitor its own interactions, we make a complete check of the left branch of the tree. In the same way, by projecting onto {`right_monitor`} and allowing `right_monitor` to monitor its own interactions, we make a complete check of the right branch. Projecting onto {`plan_monitor`} in this case would be useless, as interactions with this agent are already checked by the left and right monitors and the `plan_monitor` does not perform further checks; in particular, it does not check that messages from the left and right monitor arrive in some specific order. However, projecting onto {`plan_monitor`} would make sense if the MAS were a "sub-MAS" of a larger system, where more couples of robots exist. In that case, we might expect that each plan monitor would report the outcome of activities of its couple of robots to an agent higher in the hierarchy. Interactions with this

top-level agent should be monitored by the plan monitor (or vice-versa) and should be transparent to the agents monitoring the robots.

In the MAS implementing the ABP3 protocol shown in Figure 2, things are different due to the constraints in the fork. Although interactions induce a tree like in the SOCKS case, projecting onto Alice, Carol and Dave and allowing these three agents to check their own interactions would not be enough to verify all the protocol's constraints, as already observed in Section 4. The ABP3 cannot be distributed, hence we need a centralized monitor (which might be an external monitor or Bob himself, as it is involved in all the interactions) that "sniffs" the interactions among all the agents and verifies their compliance to the ABP3. None prevents us from projecting ABP3 also onto Alice, Carol and Dave and asking them to monitor the part of the protocol where they are involved, but this would be a useless redundancy, as Bob (or the external monitor) would already verify their part.

In order to detect the fact that, for example, projecting the ABP3 onto Dave gives no complete information on the protocol properties, we implemented an empirical method based on a "generate and test" brute force algorithm, consisting in generating all the traces of a given length of the projected protocol, and verifying if they are compliant with the global protocol. This method works only on finite traces; furthermore, while all detected positives are true, negatives may be false.

For example, Table 2 (left) shows one of the 2 different traces with length 12 of the SOCKS protocol projected onto {`right_robot, right_monitor`}. All the traces of length from 1 to 12 of the projected SOCK protocol are compliant with the global one, hence our compliance algorithms answers "maybe".

Table 2. Traces of projections of the SOCKS and ABP protocols

SOCKS protocol projected onto {right_robot, right_monitor}	ABP3 protocol projected onto {dave}
m(right_r, right_m, put_shoe)	msg(bob, dave, tell, m3)
m(right_m, right_r,	msg(dave, bob, tell, a3)
oblige_remove_shoe)	msg(bob, dave, tell, m3)
m(right_r, right_m,	msg(dave, bob, tell, a3)
removed_shoe)	msg(bob, dave, tell, m3)
m(right_r, right_m, put_shoe)	msg(dave, bob, tell, a3)
m(right_m, right_r,	msg(bob, dave, tell, m3)
oblige_remove_shoe)	msg(dave, bob, tell, a3)
m(right_r, right_m,	msg(bob, dave, tell, m3)
removed_shoe)	msg(dave, bob, tell, a3)
m(right_r, right_m, put_shoe)	msg(bob, dave, tell, m3)
m(right_m, right_r,	msg(dave, bob, tell, a3)
oblige_remove_shoe)	msg(bob, dave, tell, m3)
m(right_r, right_m,	msg(dave, bob, tell, a3)
removed_shoe)	msg(bob, dave, tell, m3)
m(right_r, right_m, put_sock)	msg(dave, bob, tell, a3)
m(right_r, right_m, put_shoe)	
m(right_m, plan_monitor, ok)	

Table 2 (right) shows the only trace with length 16 of the ABP3 protocol projected onto {dave}. This trace, as well as the shorter ones, is not compliant with the global ABP3 protocol because it does not respect the constraint that m3 must follow m1 and m2. The compliance algorithm answers "no", meaning that when projecting the ABP3 onto Dave we are no longer able to check the verification of some constraints in the global protocol.

As we have seen in Section 2, tackling the compliance problem in a formal way is a complex task, which can be faced following different approaches and heavily depends on the formalism employed for specifying protocols. Despite this interesting theoretical open problem, the compliance algorithm we have developed has proved to work well in practice in the case studies we considered.

Runtime verification of actual interactions in Jason and JADE. In our previous papers we discussed many experiments of the verification mechanism carried out on both in Jason [3] and JADE [8]. Although those experiments did not deal with projected types since projection had not been implemented yet, verifying the compliance of a set of agents w.r.t. a constrained global type works in the same way whether the type is a centralized or projected. In this paragraph we limit ourselves to briefly discussing the "socks and shoes" MAS in Jason.

The MAS is represented in Figure 1. We projected the SOCKS constrained global type shown in Section 3 onto the three sets of agents {left_monitor}, {right_monitor} and {plan_monitor}. The three resulting constrained global types are used by agents left_monitor, right_monitor and plan_monitor respectively. Each of these agents monitors all the messages that it either receives or sends, using the "message sniffing" mechanism described in [3]. We run different experiments by changing the actual messages sent by the agents in the MAS, in order to obtain both correct and wrong executions. As an example, Figure 5 shows the output of an interaction where the right_monitor sends a message with content very_good to the plan_monitor, instead of the ok content foreseen by the protocol. The plan_monitor correctly detects a dynamic type checking error (last lines of the messages in the screenshot).

Similar experiments have been carried out with JADE; the outcome of the monitoring activity in both Jason and JADE were the expected ones, both in case of correct and wrong executions.

6 Conclusions and Future Work

In this paper we have defined an algorithm for projecting a constrained global type onto a set of agents *Ags*, to allow distributed dynamic verification of the compliance of a MAS to a protocol. Besides describing the algorithm and its SWI Prolog implementation, we have framed it into the context of a full monitoring framework for agent systems, currently interfaced with Jason and JADE.

For what concerns future work, we are planning to extend the projection algorithm in order to be able to properly deal with the more general notion of attribute global type.

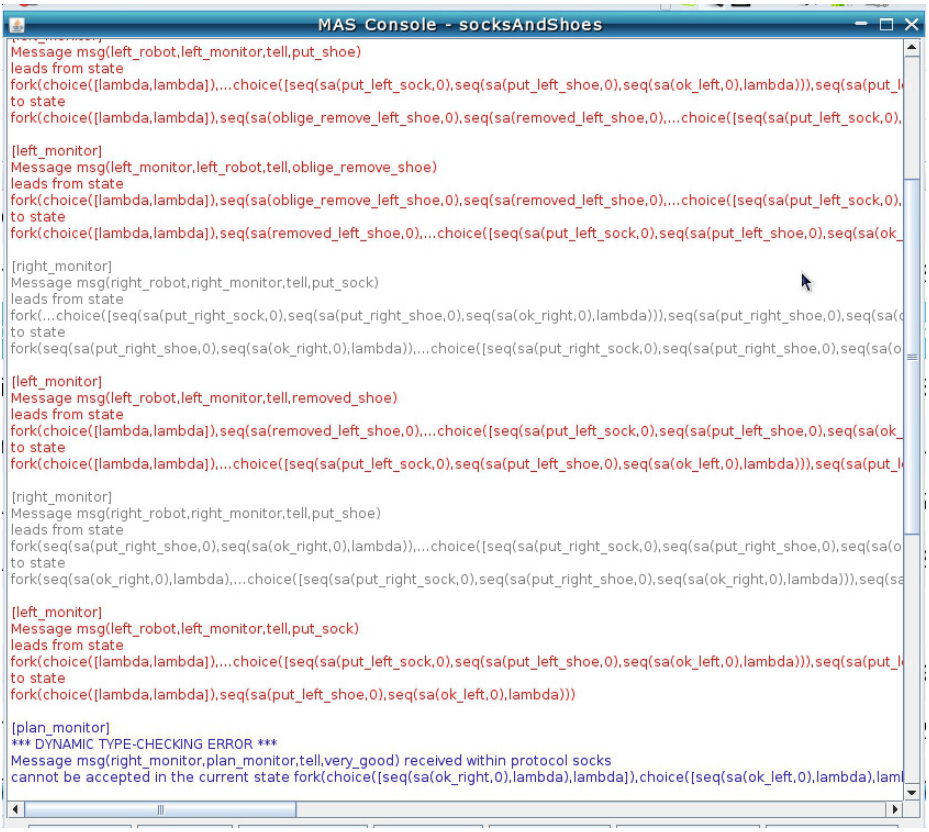

Fig. 5. Projected SOCKS protocol in Jason: the `right_monitor` violates the protocol

Also, we are investigating the possible ways to partition the set of agents for projecting types, to minimize the number of monitors, while ensuring safety of dynamic verification. In Section 2 we analyzed many different research areas, looking for solutions to the problem and for formal demonstrations that the distribution of the protocol allows monitoring the same properties as the centralized version, but even the works which seem closer to ours, namely those related with global and session types, and with choreographies, cannot be directly adopted to guarantee the correctness of the projection in our context, for four main reasons:

1. we may project on subsets of agents, if needed, and non necessarily onto individual agents;
2. we project constrained global types into constrained global types, not into "implementations": the implementation of the agents is relevant neither for the projection stage, nor for the monitoring one;
3. the expressive power of our formalism is different from other approaches: a compliance analysis must take the specific features of the formalism into account;

4. all the proposals found in literature to solve the problem of checking the correctness of projection, simply enforce syntactic restrictions on protocol specifications (as done in Scribble), whereas we would like to come out with a less restrictive approach.

While taking inspiration from these approaches will be extremely useful, we will nevertheless need to develop a new approach, taking the features and the intended use of our formalism into account.

Finally, in the examples considered in this paper, types are projected statically (that is, before the system is started) because we have assumed that agents cannot move among nodes, but monitoring would be also possible in the presence of agent mobility, as described in the scenario outlined in the introduction. However, in this case the implementation of a self-monitoring MAS is more challenging, because monitor agents have to dynamically project the global type in reaction to any change involving the set of monitored agents. Tackling scenarios of this kind is the final long term goal of our research.

References

1. Ancona, D.: Regular corecursion in Prolog. Computer Languages, Systems & Structures 39(4), 142–162 (2013)
2. Ancona, D., Barbieri, M., Mascardi, V.: Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In: Shin, S.Y., Maldonado, J.C. (eds.) Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC 2013, pp. 1377–1379 (2013)
3. Ancona, D., Drossopoulou, S., Mascardi, V.: Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason. In: Baldoni, M., Dennis, L., Mascardi, V., Vasconcelos, W. (eds.) DALT 2012. LNCS, vol. 7784, pp. 76–95. Springer, Heidelberg (2013)
4. Baldoni, M., Baroglio, C., Capuzzimati, F.: 2COMM: A commitment-based MAS architecture. In: Cossentino, M., El Fallah Seghrouchni, A., Winikoff, M. (eds.) EMAS 2013. LNCS (LNAI), vol. 8245, pp. 38–57. Springer, Heidelberg (2013)
5. Baldoni, M., Baroglio, C., Chopra, A.K., Desai, N., Patti, V., Singh, M.P.: Choice, interoperability, and conformance in interaction protocols and service choreographies. In: Sierra, C., Castelfranchi, C., Decker, K.S., Sichman, J.S. (eds.) 8th International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009, vol. 2, pp. 843–850. IFAAMAS (2009)
6. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
7. Bravetti, M., Zavattaro, G.: Contract compliance and choreography conformance in the presence of message queues. In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 37–54. Springer, Heidelberg (2009)
8. Briola, D., Mascardi, V., Ancona, D.: Distributed runtime verification of JADE multiagent systems. In: Camacho, D., Braubach, L., Venticinque, S., Badica, C. (eds.) Intelligent Distributed Computing VIII. SCI, vol. 570, pp. 81–92. Springer, Heidelberg (2014)

9. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
10. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty session. Logical Methods in Computer Science 8(1) (2012)
11. Chen, T.: Lightening global types. In: Donaldson, A.F., Vasconcelos, V.T. (eds.) Proceedings 7th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2014. EPTCS, vol. 155, pp. 38–46 (2014)
12. Chopra, A.K., Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Reasoning about agents and protocols via goals and commitments. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, vol. 1, pp. 457–464. IFAAMAS, Richland (2010)
13. Coppo, M., Dezani-Ciancaglini, M., Venneri, B.: Self-adaptive monitors for multiparty sessions. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, pp. 688–696. IEEE (2014)
14. Drusinsky, D., Shing, M.-T.: Verifying distributed protocols using MSC-assertions, run-time monitoring, and automatic test generation. In: Proceedings of the 18th IEEE/IFIP International Workshop on Rapid System Prototyping, RSP 2007, pp. 82–88 (May 2007)
15. German, E., Sheremetov, L.B.: An agent framework for processing FIPA-ACL messages based on interaction models. In: Luck, M., Padgham, L. (eds.) AOSE 2007. LNCS, vol. 4951, pp. 88–102. Springer, Heidelberg (2008)
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284. ACM (2008)
17. Krüger, I.H., Meisinger, M., Menarini, M.: Runtime verification of interactions: From MSCs to aspects. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 63–74. Springer, Heidelberg (2007)
18. Lam, S., Shankar, A.U.: Protocol verification via projections. IEEE Transactions on Software Engineering SE-10(4), 325–342 (1984)
19. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between interaction- and process-oriented choreographies. In: Cerone, A., Gruner, S. (eds.) Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, pp. 323–332. IEEE Computer Society (2008)
20. Mascardi, V., Ancona, D.: Attribute global types for dynamic checking of protocols in logic-based multiagent systems. Theory and Practice of Logic Programming, 13(4-5-Online-Supplement) (2013)
21. Mascardi, V., Briola, D., Ancona, D.: On the expressiveness of attribute global types: The formalization of a real multiagent system protocol. In: Baldoni, M., Baroglio, C., Boella, G., Micalizio, R. (eds.) AI*IA 2013. LNCS (LNAI), vol. 8249, pp. 300–311. Springer, Heidelberg (2013)
22. Meron, D., Mermet, B.: A tool architecture to verify properties of multiagent system at runtime. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2006. LNCS (LNAI), vol. 4411, pp. 201–216. Springer, Heidelberg (2007)
23. Modgil, S., Faci, N., Meneguzzi, F., Oren, N., Miles, S., Luck, M.: A framework for monitoring agent-based normative systems. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009, vol. 1, pp. 153–160. IFAAMAS, Richland (2009)

24. Neykova, R., Yoshida, N., Hu, R.: SPY: Local verification of global protocols. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 358–363. Springer, Heidelberg (2013)
25. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the theoretical foundation of choreography. In: Proceedings of the 16th International Conference on World Wide Web, WWW 2007, pp. 973–982. ACM, New York (2007)
26. Sangiorgi, D.: On the origins of bisimulation and coinduction. ACM Trans. Program. Lang. Syst. (2009)
27. Winikoff, M.: Implementing flexible and robust agent interactions using distributed commitment machines. Multiagent and Grid Systems 2(4), 365–381 (2006)
28. Wörn, H., Längle, T., Albert, M., Kazi, A., Brighenti, A., Seijo, S.R., Senior, C., Bobi, M.A.S., Collado, J.: DIAMOND: Distributed multi-agent architecture for monitoring and diagnosis. Production Planning & Control 15(2), 189–200 (2004)
29. Yolum, P., Singh, M.P.: Commitment machines. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 235–247. Springer, Heidelberg (2002)