Exploiting Prolog for Projecting Agent Interaction Protocols*

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 demonstrated that they can be used to represent complex protocols in a very compact way, and we exploited them to dynamically verify correct implementation of a protocol in a real MAS framework, Jason. The main drawback of our previous approach is the full centralization of the monitoring activity which is delegated to a unique monitor agent. 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 that can be safely used for verifying the compliance of the sub-system *Ags* to the protocol specified by the original constrained global type. The projection mechanism is implemented in SWI Prolog and 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. We leave for further investigation the problem of finding suitable partitions of agents in a MAS, to guarantee that verification through projected types and distributed agents is equivalent to verification performed by a single centralized monitor with a unique global type.

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

* The long version of this paper appears in the informal proceedings of the Second International Workshop on Engineering Multi-Agent Systems (EMAS 2014) with title "Efficient Verification of MASs with Projections". This is the shortened version presented at CILC 2014.

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 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.

In order to distribute the monitoring activity, the first step to face is to distribute the specification of the global interaction protocol in such a way that a subset of agents can monitor a subset of the interactions, still respecting the constraints stated by the global protocol.

In this paper, we address this first step by defining and implementing an algorithm for projecting the protocol representation onto subsets of agents, and then allowing interactions taking place within these subsets to be monitored by local monitors. Automatically identifying these subsets of agents in order to guarantee that the distributed monitoring behaves like the centralized one goes beyond the aims of this paper, but is matter of our current research activity.

Another 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".

The formalism that we exploit for representing and dynamically verifying agent interaction protocols, is constrained global types [2]. Global types [4] are behavioral types for specifying and verifying multiparty interactions between distributed components. We took inspiration from global types to propose "constrained global types", suitable for representing agent interaction protocols. They are based on *interactions*, namely communicative events between two agents; *interaction types*, modeling the message pattern expected at a certain point of the conversation; *producers and consumers* which allow us to express constrained shuffle of interaction traces. On top of these components, type constructors are used to model sequences, choices, concatenation and shuffle of protocols.

In our recent research we demonstrated that constrained global types can be used to represent complex protocols in a very compact way, and we exploited them to detect deviations from the protocol in a real MAS framework based on logic programming, Jason [1], and in the Java-based JADE framework[1], thanks to a bidirectional Java-Prolog interface [3]. Extensions of the original formalism with attributes have been described [5] and exploited to model a complex, real

[1] http://jade.tilab.com/.

protocol in the railway domain [6]. The integration of these 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 is always possible, this does not mean that it is always useful: as an example, the Alternating Bit Protocol discussed in this paper can be projected onto any individual agent in the MAS, but needs to be monitored in a centralized way to verify all its constraints.

The paper is organized in the following way: Section 2 briefly overviews the state of the art in distributing the monitoring activity of complex systems; Section 3 gives the technical background on constrained global types needed for presenting the projection algorithm in Section 4, Section 5 describes the implementation of the algorithm in SWI Prolog, Section 6 describes the algorithm at work, and Section 7 concludes.

2 State of the art

Many frameworks and formalism for monitoring the runtime execution of a distributed system have been proposed in the last years.

One of the most recent and relevant works in this area is SPY (Session Python) [7], a tool chain for runtime verification of distributed Python programs against Scribble (http://www.scribble.org) protocol specifications. Scribble is a language to describe application-level protocols among communicating systems initially proposed by Kohei Honda. Given a Scribble specification of a global protocol, the SPY 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 the local trace of communication actions executed during the session. This work shares the same motivations and approach with our work, and like our work concentrates on the projection of the global type to the local one rather than on the criteria for identifying in an automatic way how to distribute the monitoring activity. The main differences lie in the expressive power of the two languages, which is higher for 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, but with no emphasis on the projection from global to local monitors. This represents the main difference between those proposals and ours; the long version of this paper provides a detailed overview of many recent ones.

3 Background

This section briefly recaps on constrained global types, omitting their extension with attributes [5] 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 elements:

Interactions[2]. 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 in the set represented by τ. The superscript[3] 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.

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

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

– $\tau_1|\tau_2$ (*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.

Constrained global types are regular terms, that is, can be cyclic (recursive), and they can be represented by a finite set of syntactic equations. We limited our investigation to types that have good computational properties, namely *contractiveness* and *determinism*.

Since constrained global types are interpreted coinductively, 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) Prolog 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 only valid interaction trace respecting this constrained global type is the infinite sequence `ping pong ping pong ping pong ...`. The valid traces for the type

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

instead, are {ϵ, `ping pong`, `ping pong ping pong`, ...}, namley 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

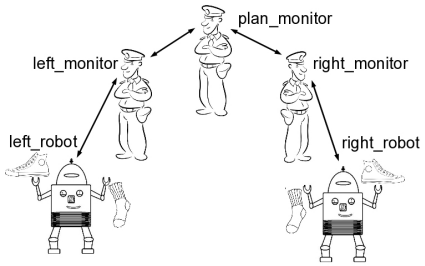

Fig. 1. The "socks and shoes" MAS

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 more crazy... 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.

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 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

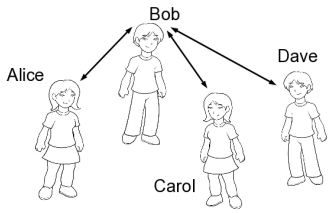

Fig. 2. The ABP3 MAS

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.

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 and acknowledges that involve it (Dave cannot be aware of the order among messages coming from other agents). 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.

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; as a first attempt, the function could be coinductively defined 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 $op \in \{+, |, \cdot\}$.

We have to consider the greatest fixed point (coinductive interpretation) of the recursive definition above, since the least fixed point (inductive interpretation) would only include non cyclic types (that is, non recursive types).

Let us consider a simple non recursive term T defined by $T = a : b : \lambda$. We want to project T on Ags. Suppose for that $involves(a, Ags)$ holds, whereas $involves(b, Ags)$ does not, 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 $\Pi(T, Ags) = a : \lambda$.

Fig. 3. Projection of recursive types.

Let us now consider the recursive type T s.t. $T = a : T'$ and $T' = b : T$. Again, the projection is driven by the syntax of T; from the definition above 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 case we can conclude by applying the base case corresponding to the λ type, in this case we do not have any basis, but we can conclude by coinduction that $\Pi(a : T', Ags)$ has to return the unique recursive type T'' s.t. $T'' = a : T''$ (see lhs picture in Figure 3).

The definition above however needs to be refined because it does not always specify a unique result for Π; to see that, let us consider the recursive type T s.t. $T = a : T'$ and $T' = b : T'$. 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', while the expected correct type should be λ, so that $\Pi(a : T', Ags) = a : \lambda$ (see rhs picture in Figure 3).

Finally, let us consider the recursive type 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 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 4 shows that type T' s.t. $T' = (a : T') + T'$ is not contractive, since the rhs cycle contains only the "+" operator. The notion of

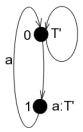

Fig. 4. Non-contractive type $T' = (a : T') + T'$

contractive type is crucial for implementing efficient runtime verification.

To ensure 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 along a path; each type is associated with its depth, 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 then *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 $T = (a : T) + (b : T)$; when computing its projection, the depth of T is 0, and initially *DeepestSeq* contains the value -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 then *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 then *DeepestSeq*, the projection of the rhs is λ.

5 Implementation

The projection algorithm has been implemented in SWI Prolog, http://www.swi-prolog.org/, which manages infinite (cyclic, recursive) terms in an efficient way. Since we need to record the association between any type and its projection in order to correctly detect and manage cycles, we exploited the SWI Prolog library `assoc` for association lists, http://www.swi-prolog.org/pldoc/man?section=assoc. Elements of an association list have 2 components: a (unique) key and a value. Keys should be ground, values need not be. An association list can be used to

fetch elements via their keys and to enumerate its elements in ascending order of their keys. The `library(assoc)` module uses AVL trees to implement association lists which makes inserting, changing and fetching a single element an O(log(N)) operation. 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` with six arguments, which are the same as those of the main predicate plus

- 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 as 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 of `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 as it has an integer number `N` associated with it. The projection is identical to the previous case, apart from the fact that `ProjT=((IntType,N):ProjT1)` in the first branch of the condition in the clause's body.
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

normalization step and hence some types which have been projected into `lambda` and might be removed, are instead kept.

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`. 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.

6 Projection at Work

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 both the complete protocols and the projected ones; also with projected types, the generated traces are correct w.r.t. the protocol specification.

For example, Table 1 (top left) shows one of the 16380 different traces with length 12 of the SOCKS protocol and Table 1 (top right) shows one of the 2 different traces with length 12 of the SOCKS protocol projected onto {`right_robot, right_monitor`} (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). Both traces correspond 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 (bottom left) shows an excerpt of one of the 30713 different traces with length 16 of the ABP3 protocol and Table 1 (bottom right) shows the first 12 interactions of the only trace with length 16 of the ABP3 protocol projected onto {`dave`}. Since the ABP3 is an infinite protocol, both traces are prefixes of infinite ones.

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 correctness, but in our experience it was enough to early detect flaws in the protocol specification.

We have implemented 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.

SOCKS protocol	SOCKS protocol projected onto {right_robot, right_monitor}
m(right_r, right_m, put_sock)	m(right_r, right_m, put_shoe)
m(left_r, left_m, put_shoe)	m(right_m, right_r, oblige_remove_shoe)
m(left_m, left_r, oblige_remove_shoe)	m(right_r, right_m, removed_shoe)
m(left_robot, left_m, removed_shoe)	m(right_r, right_m, put_shoe)
m(right_r, right_m, put_shoe)	m(right_m, right_r, oblige_remove_shoe)
m(right_m, plan_monitor, ok)	m(right_r, right_m, removed_shoe)
m(left_robot, left_m, put_shoe)	m(right_r, right_m, put_shoe)
m(left_m, left_r, oblige_remove_shoe)	m(right_m, right_r, oblige_remove_shoe)
m(left_r, left_m, removed_shoe)	m(right_r, right_m, removed_shoe)
m(left_r, left_m, put_sock)	m(right_r, right_m, put_sock)
m(left_r, left_m, put_shoe)	m(right_r, right_m, put_shoe)
m(left_m, plan_monitor, ok)	m(right_m, plan_monitor, ok)

ABP3 protocol	ABP3 protocol projected onto {dave}
msg(bob, alice, tell, m1)	msg(bob, dave, tell, m3)
msg(bob, carol, tell, m2)	msg(dave, bob, tell, a3)
msg(carol, bob, tell, a2)	msg(bob, dave, tell, m3)
msg(alice, bob, tell, a1)	msg(dave, bob, tell, a3)
msg(bob, dave, tell, m3)	msg(bob, dave, tell, m3)
msg(dave, bob, tell, a3)	msg(dave, bob, tell, a3)
msg(bob, alice, tell, m1)	msg(bob, dave, tell, m3)
msg(bob, carol, tell, m2)	msg(dave, bob, tell, a3)
msg(alice, bob, tell, a1)	msg(bob, dave, tell, m3)
msg(bob, dave, tell, m3)	msg(dave, bob, tell, a3)
msg(bob, alice, tell, m1)	msg(bob, dave, tell, m3)
msg(carol, bob, tell, a2)	msg(dave, bob, tell, a3)

Table 1. Examples of traces compliant with complete and projected protocols.

Each of these agents monitors all the messages that it either receives or sends, using the "message sniffing" mechanism described in [1].

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. All our experiments gave the expected outcome. As an example, Figure 5 shows an interaction where `left_robot` sends a `put_boot` message instead of `put_shoe`, which is correctly identified by the `left_monitor` as a violation. The conversation between the other agents goes on.

7 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. This is important in communication-intensive and highly-distributed large MASs, where a centralized approach with a unique monitoring agent would be unfeasible.

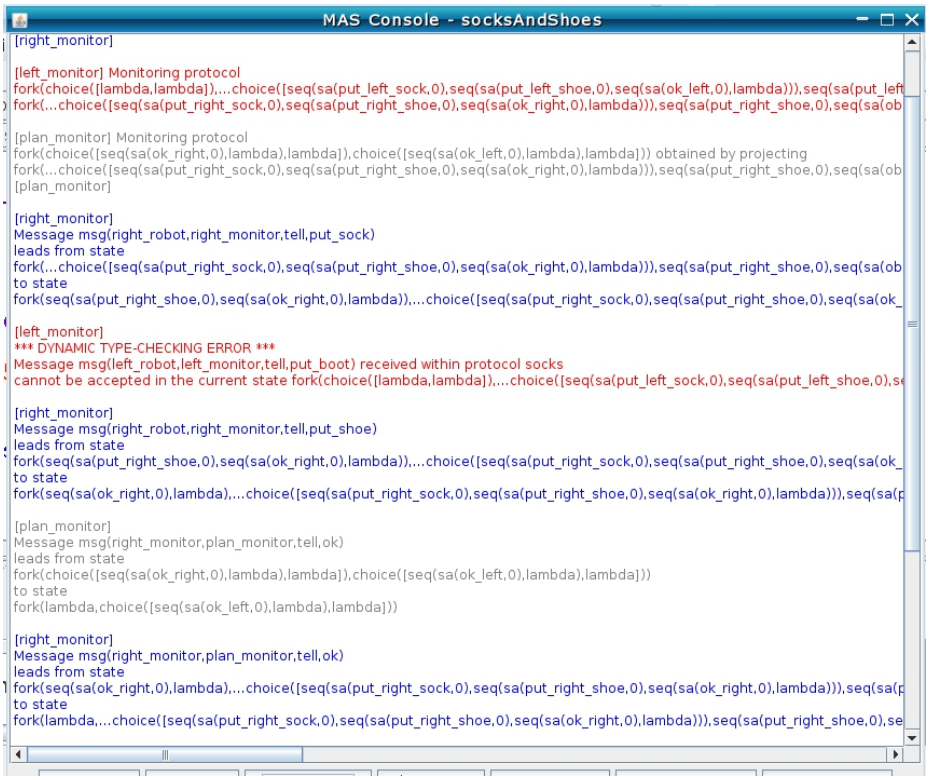

Fig. 5. The `left_robot` violates the protocol.

Besides describing the algorithm and its SWI Prolog implementation, we have shown some preliminary experiments in Jason with the running example "socks and shoes" where two local monitors with projected types are sufficient for verifying the whole system.

For what concerns future work, we are investigating on the possible ways to partition the set of agents for projecting types, to minimize the number of monitors, while ensuring safety of dynamic verification.

We are also planning to extend the projection algorithm in order to be able to properly deal with a more general form of type: attribute global types.

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 between nodes, but monitoring would be also possible in the presence of agent mobility. 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 goal of our research.

References

1. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In *DALT X*, volume 7784 of *LNAI*. Springer, 2012.
2. D. Ancona, V. Mascardi, and M. Barbieri. Global types for dynamic checking of protocol conformance of multi-agent systems. Technical report, University of Genova, DIBRIS, 2013. Extended version of *D. Ancona, M. Barbieri, and V. Mascardi. Global Types for Dynamic Checking of Protocol Conformance of Multi-Agent Systems (Extended Abstract). In P. Massazza, editor, ICTCS 2012, pp. 39-43, 2012.*
3. D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE multiagent systems. In *IDC*, Studies in Computational Intelligence. Springer, 2014.
4. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP*, LNCS, pages 2–17. Springer, 2007.
5. V. Mascardi and D. Ancona. Attribute global types for dynamic checking of protocols in logic-based multiagent systems. *TPLP*, 13(4-5-Online-Supplement), 2013.
6. V. Mascardi, D. Briola, and D. Ancona. On the expressiveness of attribute global types: The formalization of a real multiagent system protocol. In *AI*IA*, 2013.
7. R. Neykova, N. Yoshida, and R. Hu. SPY: Local verification of global protocols. In A. Legay and S. Bensalem, editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 358–363. Springer Berlin Heidelberg, 2013.