On the Expressiveness of Attribute Global Types: the Formalization of a Real Multiagent System Protocol

Viviana Mascardi, Daniela Briola, and Davide Ancona

DIBRIS, University of Genova, Italy
{viviana.mascardi,daniela.briola,davide.ancona}@unige.it

Abstract. Attribute global types are a formalism for specifying and dynamically verifying multi-party agents interaction protocols. They allow the multiagent system designer to easily express synchronization constraints among protocol branches and global constraints on sub-sequences of the allowed protocol traces.
FYPA (Find Your Path, Agent!) is a multiagent system implemented in Jade currently being used by Ansaldo STS for allocating platforms and tracks to trains inside Italian stations. Since information on the station topology and on the current resource allocation is fully distributed, FYPA involves complex negotiation among agents to find a solution in quasi-real time.
In this paper we describe the FYPA protocol using both AUML and attribute global types, showing that the second formalism is more concise than the first, besides being unambiguous and amenable for formal reasoning. Thanks to the Prolog implementation of the transition function defining the attribute global type semantic, we are able to generate a large number of protocol traces, and to manually inspect a subset of them to empirically validate that the protocol's formalization is correct. The integration of the Prolog verification mechanism into a Jade monitoring agent extending the Sniffer Agent is under way and will be used to verify the compliance of the actual conversation with the protocol.
Keywords: multiagent systems, attribute global types, negotiation, dynamic verification of protocol compliance.

1 Introduction

Railway dispatching is characterized by being physically distributed and time constrained, and by managing autonomous entities: it has been traditionally modeled using classical technologies such as operational research and constraint programming which are suitable to model static situations with complete information and a predefined number of actors, while they lack the ability to cope with the dynamics and uncertainty of real railway traffic management. Instead, Multiagent Systems (MASs, [13]) suite very well industrial scenarios where integrating and coordinating autonomous and heterogeneous entities, possibly in real time, is crucial.

For example the work presented in [3] by Böcker et al. uses a multiagent approach to the scheduling system of trains. In [1], Siahvashi and Moaveni combine the Voronoi concept of cooperative systems theory with multiagent control theory by using fuzzy control logic, and in [18] Tsang, Ho and Ip exploit software agents to coordinate train schedules at an interchange station. To model railway access negotiation, Tsang et al. [17] employ a multiagent approach in which a train services provider (TSP) and an infrastructure provider (IP) are represented by individual software agents. An IP-TSP transaction is simulated by a negotiation protocol. For flexible trains traffic control and conflict avoidance, Proenca and Oliveira [16] present a multiagent railway control system made up of two subsystems: the control one, responsible for traffic management and guidance in the network, and the learning one, that analyzes the situation descriptions accumulated in the past and infers rules that anticipate trains conflicts and improve traffic-control processes. In [10], Ghosh and Dutta model the Indian Railways as a MAS, presenting in detail the negotiation algorithm between trains with a description of the delay optimization based on the MAX-SUM algorithm.

Since MASs are open and highly dynamic, ensuring that the agents' actual behavior conforms to a given interaction protocol is of paramount importance to guarantee the participants' interoperability and security. To specify and verify multiparty interactions between distributed components, global types [8, 9, 11], a behavioral type and process algebra approach, can be profitably exploited. In [2] the problem of run-time verification of the conformance of a MAS implementation to a specified protocol was tackled by the authors by exploiting "plain" global types: they were represented as cyclic Prolog terms, on top of the Jason agent oriented programming language [4], and the transition rules were translated into Prolog too. In [15] the authors further extend the formalism by adding attributes to both sending action types and constrained global types, thus obtaining "attribute global types". This proposal was mainly inspired by a formal way to define attributes for the productions of a formal grammar, associating these attributes with values, called "attribute grammars" [14].

In this paper we test the expressiveness of attribute global types by formalizing the FYPA protocol, developed with and currently being used by Ansaldo STS, that norms the interaction among agents in a real MAS for allocating platforms and tracks to trains inside stations in Italy.

The paper is organized in the following way: Section 2 introduces attribute global types, Section 3 describes the FYPA protocol using AUML and natural language, Section 4 formalizes it using attribute global types and Section 5 concludes.

2 Attribute Global Types

The building blocks of attribute global types are the following:

Sending actions. A sending action a is a communicative event taking place between two agents.

Sending action types. Sending actions types model the message pattern expected at a certain point of the conversation. A sending action type α is a predicate on sending actions.

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

Constrained global types. A constrained global type τ represents a set of possibly infinite sequences of sending actions, and is defined on top of the following type constructors:

- λ (empty sequence), representing the singleton set $\{\epsilon\}$ containing the empty sequence ϵ.
- $\alpha^n{:}\tau$ (*seq-prod*), representing the set of all sequences whose first element is a sending action a matching type α ($a \in \alpha$), and the remaining part is a sequence in the set represented by τ. The superscript n specifies the number n of corresponding consumers that coincide with the same sending action 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 sending action a matching type α ($a \in \alpha$).
- $\tau_1 + \tau_2$ (*choice*), representing the union of the sequences of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*fork*), representing the set obtained by shuffling the sequences in τ_1 with the sequences in τ_2.
- $\tau_1 \cdot \tau_2$ (*concat*), representing the set of sequences obtained by concatenating the sequences of τ_1 with those of τ_2.
- The "meta-construct" *fc* (for *finite composition*) that takes τ, a constructor *cn*, and a positive natural number n as inputs and generates the "normal" constrained global type (τ *cn* τ *cn* ... *cn* τ *cn* λ) (n times).

Attribute global types are constrained global types whose sending action types may have attributes, included within curly brackets, and that have been extended to provide contextual information by means of further attributes and conditions, included in square brackets.

Attribute 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.* The semantics of attribute global types has been provided in [15].

3 The FYPA protocol

3.1 The problem and the model

The FYPA (Find Your Path, Agent!) multiagent system [5–7] was developed starting from 2009 by the Department of Informatics, Bioengineering, Robotics and System Engineering of Genoa University and Ansaldo STS, the Italian leader in design and construction of signalling and automation systems for conventional and high speed railway lines. It aims at automatically allocating trains moving into a railway station to tracks, in order to allow them to enter the station and reach their destination (either the station's exit or a node where they will stop) considering real time information on the traffic inside the station and on availability of tracks. The problem consists of:

– A set of indivisible resources (tracks in a railway station) that must be assigned to different entities (trains) in different time slots (each resource can be used by only one entity in each time slot, and we use a linear and discrete model of time).

– A set of entities (trains) with different priorities, each needing to use some resources for one or more time slots.

– A mixed multi-graph of dependencies among resources: an entity can start using resource R only if it used exactly one resource from $\{R_1, R_2, ..., R_n\}$ in the previous time slot (we represent these dependencies as arcs $R_1 \rightarrow R$, $R_2 \rightarrow R$, ..., $R_n \rightarrow R$ in the graph). There may be many different direct arcs connecting the same two resources, and arcs may be both direct and bidirectional.

– A set of couples of conflicting or bidirectional arcs in the graph of dependencies: for presentation purposes, in this paper we discard these kinds of arcs.

– A static allocation plan that assigns resources to entities (namely, a path in the graph) for pre-defined time slots, in such a way that no conflicts arise.

In an ideal world where resources never go out of order and where any entity in the system can always access the resources assigned to it by the static allocation plan, no problems arise. In the real world where entities happen to use resources for longer than planned and where resources can break up, a dynamic re-allocation of resources over time is often required. Thus, the solution of the real world problem is a dynamic re-allocation of the resources to the entities such that: 1. the re-allocation is feasible, namely free of conflicts; in our scenario, conflicts may arise because two or more entities would want to access the same resource in the same time slot; 2. the re-allocation task is completed within a pre-defined amount of time; 3. each entity minimizes the changes between its new plan and its static allocation plan; 4. each entity minimizes the delay in which it reaches the end point of the path with respect to its static allocation plan; 5. the number of entities and resources involved in the re-allocation process is kept to the minimum.

Considering the requirements of Ansaldo STS's application to find a solution in quasi-real time, the plan resulting from the re-allocation problem may be sub-optimal.

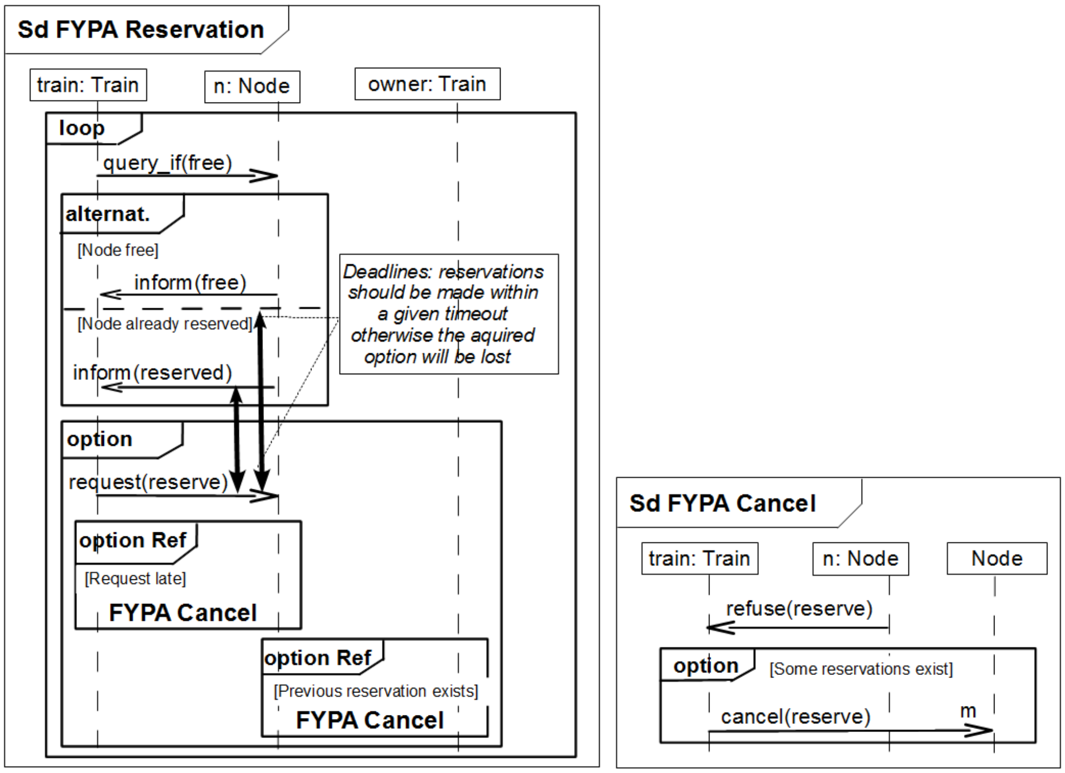

Fig. 1. FYPA Reservation protocol with its FYPA Cancel sub-protocol.

3.2 FYPA Protocol Description

The FYPA Reservation protocol is shown, expressed in the AUML formalism [12], in Figure 1. Messages have been substituted by short labels for readability, and the Cancel sub-protocol has been described separately. The description of the FYPA protocol in AUML is an original contribution of this paper, as the previous publications described it in a very informal and shallow way.

The FYPA Reservation protocol involves agents representing trains ("Train" role in the AUML diagram), and agents managing nodes ("Node" role).

Each train knows the paths $\{P_1 = N_{from}...N_{to}; ...; P_k = N_{from}...N_{to}\}$ it could follow to go from the node where it is (N_{from}), to the node where it needs to stop (N_{to}). Such paths are computed by a legacy Ansaldo application, which is wrapped by an agent named *PathAgent* and is accesses by the FYPA MAS through web services. We do not model the *PathAgent* here and we limit ourselves to describe the interactions among trains and nodes. Each train also knows which path it is currently trying to reserve, how many nodes answered to its requests and in which way, and how much delay it can still accept: to reserve a path, the train must obtain a reservation for each node in it. To reserve a node, a train must ask if it is free, wait for the answer from the node (free or already reserved by another train in an overlapping time slot) and then must reserve the resource, which might also mean stealing it to the train that reserved it before. In this case, which usually takes place if the priority of the "thief" is higher than that of the "robbed", the node will inform the robbed train, that will search for another path or for the same path in different time slots. Each node knows the arcs that it manages (those that enter in it). It also knows which trains optioned or reserved the node, in which time slots, from which node they are expected to arrive, and which arc they have been allowed to traverse.

The protocol control flow is specified by the AUML boxes that represent loops, alternative choices, optional choices, and references to sub-protocols. When a train fires a FYPA Reservation protocol instance by sending a request of information to a node ($query\_if(free)$ label), a conversation among them starts and a fresh conversation identifier is generated by the train to keep track of it. If the same agent sends another request of information to the same node, the started conversation is considered as a new one and a new fresh id must identify it, whereas when a train contacts more nodes that build up a path, the conversation id remains the same. We discuss the FYPA Reservation protocol from the train perspective, by expanding the message labels into the actual message types, starting from the top.

$query\_if(free)$: in order to reserve a path $P_1 = N_{from}...N_{to}$, a train *Train* sends messages of type *msg(Train,* N_i*, query_if(free(MyPr, T1, T2, From)), cid(CId))* to each node N_i in the path. *MyPr* is *Train*'s priority and can be changed into ∞ if the reservation is not disputable, *T1* and *T2* are the extremes of the time slot when *Train* would reserve the node, and *From* is the node from which it will arrive. A "non-disputable" request is issued only when *Train* has no other chances to move inside the station, and it must necessarily move on *N*.

$inform(free)$: if the node *N* that receives the request is free in the requested time slot, it answers with a message of type *msg(N, Train, inform(free(Arc,From), expires(Timeout)), cid(CId))*, where *Arc* is the arc, among the possibly many ones, that *Train* is allowed to use for moving from *From* to *N*. *N* will not answer other requests for the same or an overlapping time slot until either *Train* reserves definitely the resource or *Timeout* expires, in which case *Train* looses its option.

$inform(reserved)$: if *N* is already allocated to another train in the requested time slot, it answers with a message of type *msg(N, Train, inform(reserved (Owner, OwnerPr, Arc, From, NextT3, NextT4), expires(Timeout)), cid(CId))*, where *Owner* is the train that currently owns the node, *OwnerPr* is its priority, *Arc* is the arc that *Train* could have used to reach *N* from *From*, *NextT3* and *NextT4* are the extremes of the next free slot. *OwnerPr* may assume the value ∞ to mean that the reservation cannot be stolen.

$request(reserve)$: *Train* may decide to reserve the node either if the node is free, or if it is already reserved but the priority of the current owner is lower or equal than its priority (or in other complex situations not described in this paper). In both these cases, *Train* sends a message with type *msg(Train, N, request(reserve(Arc, MyPr, Owner, OwnerPr, T1, T2), expired(AbsTime)),* *cid(CId)). AbsTime* is the absolute time when the option would have expired. We do not enter here in the details of when a train decides to steal a node to another train, that is based on the delay it could undergo and on its priority. If this happens, *N* must start a FYPA Cancel sub-protocol between *N* and *Owner*, to inform it that it lost the resource it reserved. On the other hand, it may also happen that *Train*'s reservation fails because it was issued to late w.r.t. the given timeout, firing a FYPA Cancel sub-protocol between *N* and *Train*. A train can also decide not to confirm the reservation, letting it expire.

In case the initial request of *Train* was not disputable (represented by an infinite priority of *Train*), and the node was already reserved by a train with infinite priority (meaning that the current owner's request was not disputable as well), a human intervention is required. This situation can in fact arise only when the station is undergoing a disaster, with so many failures of tracks and nodes that trains cannot move any longer. Apart from this emergency situation, not shown in the diagram for sake of clarity, the allocation of nodes and arcs to trains is entirely managed by the MAS in an automatic way.

We do not enter into the details of the FYPA Cancel sub-protocol, since it is almost intuitive and we do not describe it using attribute global types.

4 Protocol Specification Using Attribute Global Types

4.1 Constraints on the FYPA Reservation Protocol's Traces

To design and develop attribute global types, we need both the protocol's information flow, and (optionally) the constraints that the protocol traces must respect. The more constraints we are able to formalize, the more complete the verification mechanism will be.

The traces of the FYPA protocol must respect many constraints that we could not express in AUML without compromising the readability diagram, and that involve the content of both single messages and sets of messages. In this section we list some of them. All the traces shown below have been obtained by the automatic trace generation process mentioned in Section 4.2.

"Local" constraints on messages. Each message must have the right type. For example, a *request* message must be sent by an agent playing the "Train" role to an agent playing the "Node" role and the arguments of the *reserve* action must be a well formed arc identifier, the priority of the sender, the train to which the resource will be stolen ("none" is the node was free), its priority ("0" if it was free), and the extremes of the reserved time slot. Such a message is correct only if the priority of the train making the reservation is higher than or equal to the priority of the train that already owned the node, if any. These constraints can be easily verified by inspecting the arguments of the message, without needing to exploit attributes. A message like

```
msg(t1, n5, request(reserve(arc(4, 5, d), 3, t3, 1, 23, 31), ...))
```

satisfies them.

"Horizontal" constraints on queries for reserving a path. When a train contacts a sequence of nodes to verify whether they are free in order to optionally issue a reservation request, the arguments of these messages must form a coherent path: the "From" argument in message m_{i+1} must be the same as the receiver of message m_i, the time slot's first extreme in message m_{i+1} must be the same as the time slot's second extreme in message m_i, the conversation id must be the same, and the train cannot change its priority, apart from setting it equal to infinity for non disputable requests. These constraints can be verified without needing to know the station's topology. For example, the trace

```
msg(t1, n5, query_if(free(infinity, 10, 15, n4)), cid(c1))
msg(t1, n6, query_if(free(3, 15, 17, n5)), cid(c1))
msg(t1, n7, query_if(free(3, 17, 22, n6)), cid(c1))
msg(t1, n8, query_if(free(infinity, 22, 44, n7)), cid(c1))
msg(t1, exit, query_if(free(3, 44, 48, n8)), cid(c1))
```

respects them.

"Vertical" constraints on conversations between a train and a node. Apart from the requirement that during a single conversation the train changes neither its distinguishing features (such as its priority) nor the conversation features (the conversation id), we can identify some more constraints:

1. If a node is reserved, it must inform the train of the arc it could have used to reach it and of the time slot when it will be free again. This time slot must start after the time slot indicated by the train in its *query_if* message, even if it may overlap with it.

2. When a train decides to reserve a node, it must access it by the arc included in the node's answer.

A trace like

```
msg(t2, n5, query_if(free(3, 22, 44, n4)), cid(c2))
msg(n5, t2, inform(reserved(t3, 1, arc(4, 5, a), n4, 23, 31), ...))
msg(t2, n5, request(reserve(arc(4, 5, a), 3, t3, 1, 22, 44), ...))
```

respects both constraints.

Since a train can interact with the same node many times, for example because the attempt to reserve a path failed and then the train has to try to reserve a new one, we add another vertical constraint that involves conversation loops: if a train sends more than one *query_if* message to the same node, the conversation id must be different since the messages belong to different conversations. Also, if the node answered that it was not available, in the new *query_if* message the train must ask for a time slot successive to the previous one (it would be useless to ask for the same time slot, if the train already knows it is not available), unless the train changed its priority to infinity in the meanwhile.

4.2 FYPA Reservation Protocol Using Attribute Global Types

In this section we give the flavor of how the FYPA Reservation attribute global type looks like and which features of the formalism were exploited to model the constraints devised in Section 4.1. The correctness of the formalization with respect to the actual protocol has been empirically validated by automatically generating a large number of protocol traces, under different initial conditions (number of trains, station configuration), and by manually inspecting a randomly selected subset of them.

The trace generation has been carried out thanks to the implementation of the transition function $\delta{:}\mathcal{CT} \times \mathcal{A} \rightarrow \mathcal{P}_{\mathit{fin}}(\mathcal{CT})$ described in [15], where $\mathcal{CT}$ and $\mathcal{A}$ denote the set of contractive attribute global types and of sending actions, respectively. The δ function defines the interpretation of an attribute global type and has been implemented using SWI Prolog[1].

Core of the protocol. A single conversation between a train and one node in the FYPA protocol is represented by the following attribute global type:

$$\begin{array}{l}
\mathit{FYPA\_NODE\_RESERVATION} = (\\
(\mathit{ND\_QUERY} : ((\mathit{NONDISP\_NONDISP} + \mathit{WT\_DISP}) + \mathit{FREE\_NODE}))+ \\
(\mathit{D\_QUERY} : ((\mathit{DISP\_NONDISP} + \mathit{WT\_DISP}) + \mathit{FREE\_NODE})))
\end{array}$$

The train may issue either a non disputable *query_if* ($\mathit{ND\_QUERY}$) or a disputable one ($\mathit{D\_QUERY}$). In the first case, three different situations may take place: either the node has a non disputable reservation ($\mathit{NONDISP\_NONDISP}$), which leads to the involvement of a human operator, or it has a disputable reservation ($\mathit{WT\_DISP}$, for "whatever-disputable"), in which case the train may steal the node, or it is free, in which case the train may reserve the node. In case of a disputable query, the protocol is similar apart from the $\mathit{DISP\_NONDISP}$ branch. The equations defining the sub-types are given below. Despite the complexity of the protocol, whose description using AUML and natural language required many pages of this paper, its representation using attribute global types is rather compact.

$$\begin{array}{l}
\mathit{ND\_QUERY} = \mathit{msg}(\mathit{Train}, N, \mathit{nd\_query\_if}(\mathit{free}(\mathit{infinity}, T1, T2, \mathit{From})), \mathit{cid}(\mathit{CId}))^1 \\
\mathit{D\_QUERY} = \mathit{msg}(\mathit{Train}, N, \mathit{d\_query\_if}(\mathit{free}(\mathit{MyPr}, T1, T2, \mathit{From})), \mathit{cid}(\mathit{CId}))^1 \\
\mathit{CONSTR\_ON\_PATH}\{\mathit{List}, (\mathit{Train}, \mathit{MyPr}, N, T1, T2, \mathit{From}, \mathit{CId})\} = \\
((\mathit{msg}(\mathit{Train}, N, \mathit{nd\_query\_if}(\mathit{free}(\mathit{MyPr}, T1, T2, \mathit{From})), \mathit{cid}(\mathit{CId})) : \\
\mathit{CONSTR\_ON\_PATH}\{[(\mathit{Train}, \mathit{MyPr}, N, T1, T2, \mathit{From}, \mathit{CId})|\mathit{List}],_{\}} +\lambda))+ \\
(\mathit{msg}(\mathit{Train}, N, \mathit{d\_query\_if}(\mathit{free}(\mathit{MyPr}, T1, T2, \mathit{From})), \mathit{cid}(\mathit{CId})) : \\
\mathit{CONSTR\_ON\_PATH}\{[(\mathit{Train}, \mathit{MyPr}, N, T1, T2, \mathit{From}, \mathit{CId})|\mathit{List}],_{\}} +\lambda))) \\
\text{[the tuple (Train, MyPr, N, T1, T2, From, CId) forms a consistent path with the elements in List]} \\
\mathit{ND\_RESERVED}\{T2, \mathit{NextT3}\} = (\mathit{msg}(N, \mathit{Train}, \mathit{inform}(\mathit{nd\_reserved}(\mathit{Ow}, \mathit{OwPr}, \mathit{Arc}, \mathit{From}, \\
\mathit{NextT3}, \mathit{NextT4}), \mathit{expires}(\_\mathit{TO})), \mathit{cid}(\mathit{CId}))^0 : \lambda)[\mathit{NextT3} > T2] \\
\mathit{D\_RESERVED}\{T2, \mathit{NextT3}\} = (\mathit{msg}(N, \mathit{Train}, \mathit{inform}(\mathit{d\_reserved}(\mathit{Ow}, \mathit{OwPr}, \mathit{Arc}, \mathit{From}, \\
\mathit{NextT3}, \_\mathit{NextT4}), \mathit{expires}(\_\mathit{TO})), \mathit{cid}(\mathit{CId}))^0 : \lambda)[\mathit{NextT3} > T2] \\
\mathit{DISASTER} = \mathit{msg}(N, \_\mathit{HumanOperator}, \mathit{inform}(\mathit{disaster}(\mathit{Ow}, \mathit{Train}, T1, T2)), \mathit{cid}(\mathit{CId}))^0 \\
\mathit{STEAL\_ON\_TIME} = (\mathit{msg}(\mathit{Train}, N, \mathit{request}(\mathit{reserve\_on\_time}(\mathit{Arc}, \mathit{MyPr}, \mathit{Ow}, \mathit{OwPr}, T1, T2), \\
\mathit{expired}(\mathit{AbsTime})), \mathit{cid}(\mathit{CId}))^0 : \mathit{REFUSE\_NODE\_ROBBED\_TRAIN}) \\
\mathit{STEAL\_LATE} = ((\mathit{msg}(\mathit{Train}, N, \mathit{request}(\mathit{reserve\_late}(\mathit{Arc}, \mathit{MyPr}, \mathit{Ow}, \mathit{OwPr}, T1, T2), \\
\mathit{expired}(\mathit{AbsTime})), \mathit{cid}(\mathit{CId}))^0 : (\mathit{REFUSE\_NODE\_SAME\_TRAIN} + \lambda)) + \lambda) \\
\mathit{RESERVE\_ON\_TIME} = (\mathit{msg}(\mathit{Train}, N, \mathit{request}(\mathit{reserve\_on\_time}(\mathit{Arc}, \mathit{MyPr}, \mathit{none}, 0, T1, T2), \\
\mathit{expired}(\mathit{AbsTime})), \mathit{cid}(\mathit{CId}))^0 : \lambda) \\
\mathit{RESERVE\_LATE} = (\mathit{msg}(\mathit{Train}, N, \mathit{request}(\mathit{reserve\_late}(\mathit{Arc}, \mathit{MyPr}, \mathit{none}, 0, T1, T2), \\
\mathit{expired}(\mathit{AbsTime})), \mathit{cid}(\mathit{CId}))^0 : \mathit{REFUSE\_NODE\_SAME\_TRAIN})
\end{array}$$

[1] http://www.swi-prolog.org/.

$REFUSE\_NODE\_SAME\_TRAIN = (msg(N, Train, refuse(reserve), cid(CId))^0 : \lambda)$
$REFUSE\_NODE\_ROBBED\_TRAIN = (msg(N, Ow, refuse(reserve), cid(\_CIdX))^0 : \lambda)$
$FREE = (msg(N, Train, inform(free(Arc, From), expires(\_TO)), cid(CId))^0 : \lambda)$
$NONDISP\_NONDISP = (ND\_RESERVED \cdot (DISASTER : \lambda))$
$WT\_DISP = (D\_RESERVED\{T2, NextT3\} \cdot ((STEAL\_ON\_TIME + STEAL\_LATE) + \lambda))$
$DISP\_NONDISP = ND\_RESERVED\{T2, NextT3\}$
$FREE\_NODE = (FREE \cdot ((RESERVE\_ON\_TIME + RESERVE\_LATE) + \lambda))$

For each message type appearing in the equations, the specification of which actual messages have that type and under which constraints the type is correct, must be provided. To make an example, the message type appearing in $STEAL\_ON\_TIME$ holds on messages $msg(Train, N, request(reserve(Arc, MyPr, Owner, OwnerPr, T1, T2), expired(AbsTime)), cid(CId))$ if the following conditions are satisfied:

$current\_time(CurrentTime), AbsTime >= CurrentTime, is\_arc(Arc), is\_priority(OwnerPr),$
$is\_priority(MyPr), is\_higher\_priority(MyPr, OwnerPr), role(Train, train), role(N, node),$
$role(Owner, train), different(Owner, Train), is\_conv\_id(CId), is\_time\_range((T1, T2)).$

Note that message types also allow us to distinguish between reservations that arrive on time and reservations that arrive late, just by comparing the actual current time $current\_time(CurrentTime)$ with the time when the reservation expired. This is a very powerful mechanism to model deadlines that may cause different courses of actions if they are not met.

Because of space constraints we only describe the trickiest aspects of the equations above. Given the introduction to the language syntax given in Section 2, most of them should be intuitive. All the messages apart from those in ND_QUERY and D_QUERY have 0 as superscript. This means that 0 consumers for these messages are present in the protocol, and no synchronization among fork branches will take place. This is not true for the messages in ND_QUERY and D_QUERY that will be consumed by the respective consumers in $CONSTR\_ON\_PATH$, in order to synchronize many conversations between the same train and different nodes and to ensure that the $query\_if$ contents form a consistent path. This constraint has been implemented by equipping $CONSTR\_ON\_PATH$ with attributes, namely the contents of all the $query\_if$ sent by train $Train$ to the nodes in the path, accumulated in a list which is passed from one instance of $CONSTR\_ON\_PATH$ to the successive one.

Another constraint that we expressed in Section 4.1 is that if a node answers to a $query\_if$ that it is reserved, it must also state when it will become available again, and this time slot must be successive to the one requested by the train. This is formalized by adding the two attributes $\{T2, NextT3\}$ to $D\_RESERVED$, and asking that $[NextT3 > T2]$.

Multiple conversations with different nodes to reserve a path are modeled by the following attribute global type:

$fc(FYPA\_PATH\_RESERVATION, FYPA\_NODE\_RESERVATION, |, NumOfNodes)$
$FYPA = (FYPA\_PATH\_RESERVATION | CONSTR\_ON\_PATH)$

Given $NumOfNodes$ the number of nodes that the train is expected to contact, $fc(FYPA\_PATH\_RESERVATION, FYPA\_NODE\_RESERVATION, |, NumOfNodes)$ creates an attribute global type named $FYPA\_PATH\_RESERVATION$, obtained by replicating $FYPA\_NODE\_RESERVATION$ $NumOfNodes$ times, using the fork constructor. From an implementative viewpoint, all these replicas have fresh new variables to ensure that they are independent from one another.

$(FYPA\_PATH\_RESERVATION|CONSTR\_ON\_PATH)$ defines one single iteration of the $FYPA$ protocol, where the fork is used to synchronize the two branches and guarantee that inquiries to nodes form a consistent path.

Finally, the external conversation loop modeling more iterations of the $FYPA$ protocol is represented by

$$LOOP\{N, PrevCId, CId, PrevT1, T1\} = (FYPA \cdot LOOP\{N, CId, \_, T1, \_\})$$
$$[\text{different(PrevCId,CId)} \wedge \text{different(PrevT1,T1)}]$$

5 Conclusions and Future Work

The comparison between the informal AUML representation of the protocol, that must be necessarily accompanied by an explanation in natural language, and the representation using attribute global types, which is compact, unambiguous, and amenable for formal reasoning, suggests that the second is definitely better for all those situations where on-line verification of protocol compliance is required, even if it is less intuitive and requires some training to be used.

Nowadays, a Jason version of the monitor agent able to control at runtime a conversation among agents following a specified protocol exists. However, many industrial strength MASs have been implemented in Jade, as the FYPA one. We are thus designing a Jade version of the monitor agent: this agent will integrate a Prolog engine to exploit the Prolog representation of the attribute global type describing a protocol, and the transition function to verify the compliance of actual messages to the protocol. The monitor should be able to check the ongoing execution of the MAS without interfering with the agents, so that no changes are required to the agents' code or behavior. We have already implemented a first prototype of the Jade monitor agent, that acts in a similar way as the Jade sniffer agent: when it starts, it registers itself as a sniffer to the Jade Agent Management System, so that it will be informed of any new agent creation and of all the events concerning message exchange among the agents that it sniffs. In this way when a new agent is created, if it is in the list of those to be monitored, the monitor will receive a copy of all the messages that involve such agent, checking if it is respecting the protocol.

Our close future work consists in completing the implementation of such agent and in using it to monitor the FYPA MAS.

References

1. B. M. Ali Siahvashi. Automatic train control based on the multi-agent control of cooperative systems. *TJMCS*, 1(4):247–257, 2010.
2. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason. In *DALT X. Revised, Selected and Invited Papers*, volume 7784 of *LNAI*. Springer, 2012.
3. J. Böcker, J. Lind, and B. Zirkler. Using a multi-agent approach to optimise the train coupling and sharing system. *European Journal of Operational Research*, 131(2):242 – 252, 2001.
4. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons, 2007.
5. D. Briola and V. Mascardi. Design and implementation of a NetLogo interface for the stand-alone FYPA system. In *WOA 2011, Proceedings*, pages 41–50, 2011.
6. D. Briola, V. Mascardi, and M. Martelli. Intelligent agents that monitor, diagnose and solve problems: Two success stories of industry-university collaboration. In *Journal of Information Assurance and Security*, volume 4, pages 106–117, 2009.
7. D. Briola, V. Mascardi, M. Martelli, R. Caccia, and C. Milani. Dynamic resource allocation in a MAS: A case study from the industry. In *From Objects to Agents Workshop, WOA 2009, Proceedings*, 2009.
8. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07 (part of ETAPS 2007)*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
9. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty session. *Logical Methods in Computer Science*, 8(1), 2012.
10. S. Ghosh and A. Dutta. Multi-agent based railway track management system. *In Proc. of 3rd IEEE International Conference on Advance Computing & Communication*, pages 1408–1413, 2013.
11. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL 2008*, pages 273–284. ACM, 2008.
12. M.-P. Huget, B. Bauer, J. Odell, R. Levy, P. Turci, R. Cervenka, and H. Zhu. FIPA modeling: Interaction diagrams. Working Draft Version 2003-07-02. Online at http://www.auml.org/auml/documents/ID-03-07-02.pdf, 2003.
13. N. R. Jennings, K. P. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
14. D. E. Knuth. The genesis of Attribute Grammars. In P. Deransart and M. Jourdan, editors, *WAGA'90*, volume 461 of *LNCS*, pages 1–12. Springer, 1990.
15. V. Mascardi and D. Ancona. Attribute global types for dynamic checking of protocols in logic-based multiagent systems (technical communication). To appear in Theory and Practice of Logic Programming, On-line Supplement, as technical communication of the ICLP 2013 conference, 2013.
16. H. Proença and E. Oliveira. Marcs multi-agent railway control system. In *IBERAMIA*, pages 12–21, 2004.
17. C.-W. Tsang and T.-K. Ho. Optimal track access rights allocation for agent negotiation in an open railway market. *Intelligent Transportation Systems, IEEE Transactions on*, 9(1):68–82, 2008.
18. C. W. Tsang, T. K. Ho, and K. H. Ip. Train schedule coordination at an interchange station through agent negotiation. *Transportation Science*, 45(2):258–270, May 2011.