Parametric Protocol-Driven Agents and their Integration in JADE*

Angelo Ferrando

DIBRIS, Genoa University, Italy
s3479302@studenti.unige.it

Abstract. In this paper we introduce "Template Global Types" which extend Constrained Global Types to support a more generic and modular approach to define protocols, meant as patterns of events of a given type. Protocols can be used both for monitoring the behavior of distributed computational entities and for driving it. In this paper we show the potential of Template Global Types in the domain of protocol-driven intelligent software agents. The interpreter for "executing" Template Global Types has a very natural implementation in Prolog which can easily implement the transition rules for moving from one state to another one, given that an event has been perceived (in case of monitoring) or generated for execution (in case of protocol-driven behavior). This interpreter has been integrated into the Jason logic-based agent framework with limited effort, thanks to the native support that Jason offers to Prolog. In order to demonstrate the flexibility and portability of our approach, which goes beyond the boundaries of logic-based frameworks, in this paper we discuss the integration of the protocol-driven interpreter into the JADE agent framework, entirely implemented in Java.

1 Introduction

Nowadays, having guarantees on the correct behavior of developed systems is gaining more and more importance. Especially in the case of distributed systems, increasing their robustness is a mandatory target. To achieve this goal, we can use two different techniques:

- Testing, which reduces the percentage of bugs in software by trying out the largest number of features offered by the system in order to find errors or inconsistencies.
- Verification, which allows the developers to perform an exhaustive search in order to check if all chosen properties are maintained, such as the absence of deadlock within a concurrent system.

Checking correctness is not an easy problem, especially when considering distributed systems.

* The paper contains original material. The author is a student of the Computer Science Master's Degree at Genova University.

A typical example of complex, heterogeneous, open and dynamic distributed system is the multi-agent system (MAS), where each component is autonomous and communication is vital.

In order to verify the correct behavior of the agents we can monitor them with the help of a monitor agent, as the one by Briola et al. [6]. An evolution of that approach is described by Ancona et al. [11] in which, instead of having one monitor agent that controls the behavior of the system by checking that everything is running correctly, the behavior of each individual agent is driven by the protocol. Since the agent's behavior is driven by a protocol, it is correct without needing to be controlled, or better, the agent in a certain way becomes controller of itself, being able to do only what the protocol allows it to do.

Protocols can be defined using many different formalisms. In the previous work carried out at the University of Genova, an original formalism named "Global Types" was proposed along with its successive extensions including "Constrained Global Types" [1, 2].

Protocols expressed as Constrained Global Types include both Prolog equations and transition rules moving from one allowed protocol state to another one, given a perceived (or a generated, in case of protocol-driven agents) event fully implemented in Prolog. For this reason, the first attempt to implement an interpreter for protocol-driven agents was with Jason[1], because it is able to run Prolog code directly within the agent [11].

This paper advances a previous work [11] in the following issues:

- first extends Constrained Global Types with a mechanism for making them more modular and flexible ("Template Global Types");
- second implements agents driven by protocols expressed using Template Global Types in Jason;
- third implements agents driven by protocols expressed using Template Global Types in a framework not based on Prolog. The choice fell on JADE[2], a Java platform for the creation of multi-agent systems widely used in industrial applications (see Figure 1).

The integration of an interpreter for Template Global Types protocol-driven allows us to demonstrate two important points:

- the choice of the framework for the implementation of the MASs is not a constraint. For instance, agents can follow the BDI (Beliefs, Desires, Intentions [23]) approach, as happens in Jason, or not, as happens in JADE;
- the only requirement is the ability to implement an interface between a Prolog engine supporting cyclic terms and the underlying MAS framework.

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

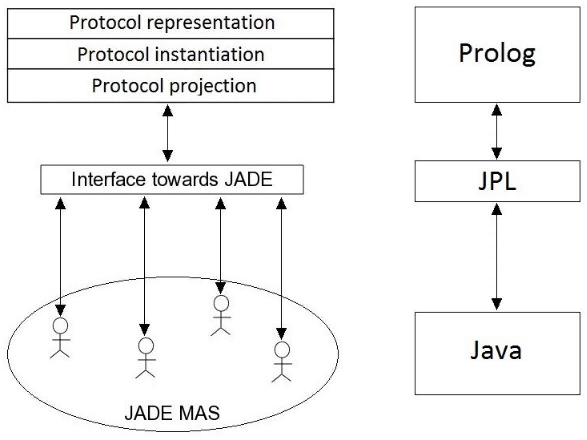

Fig. 1. Our framework for protocol-driven JADE agents.

2 Background

In [11] we used Constrained Global Types without variables (that is instead typical of Attribute Global Type [18]). In order to better understand the content of this paper, we briefly sketch their syntax and semantics.

Omitting the operators that are not used in our examples, a constrained global type may look like the following:

– *EvType:PrSpec* (*sequence*), where *EvType* is an event type and *PrSpec* is a constrained global type. From a semantic point of view, *EvType:PrSpec* represents the set of all event traces whose first event *Ev* matches the event type *EvType* ($Ev \in EvType$), and the remaining part is a trace in the set represented by *PrSpec*.
– $PrSpec_1 \mid PrSpec_2$ (*fork*), where both $PrSpec_1$ and $PrSpec_2$ are constrained global types, representing the set obtained by shuffling the traces in $PrSpec_1$ with the traces in $PrSpec_2$.

With these two operators we can already write complex protocols, as we will see afterwards with some examples.

Another important aspect that we want to introduce, and that was widely discussed in [11], is the *switch* interaction. Such an event can take place when our events include communicative ones. During an interaction protocol execution, an agent can ask to another to change its protocol; this can happen only in specific circumstances and is an important feature in order to support runtime protocol switch.

3 Template Global Types

To be able to write more complex protocols, we extended the previous work [11] introducing Template Global Types. The main difference of Template Global Types w.r.t. Constrained Global Types is the presence of parameters inside the protocol definition.

Template Global Types are a sort of "meta-formalism" in the sense they cannot be directly used as they are. Indeed, they are templates which must be applied to some arguments in order to obtain plain Constrained Global Types. Parameters are present only in the Template Global Type definition: when Template Global Types are actually used either for runtime verification or for protocol driven behavior generation, all terms must be ground, i.e. variables must have already been instantiated.

In order to better explain this new formalism, we introduce some examples.

Here is how we would have represented a client-server protocol with Constrained Global Types[3].

```
                        SERVER =
(receive_request(client1),0):(serve_request(client1),0):SERVER.
```

An example of correct trace would be

```
receive_request(client1): serve_request(client1):
          receive_request(client1):...
```

where ... indicates that the trace is infinite: Constrained Global Types use coinduction to easily represent endless sequences.

The `SERVER` protocol is a client-server protocol made up by a loop in which the server receives a `serve_request` from the client1 replying with a `receive_request`.

Global types can be easily expressed as a set of Prolog equations like the one defining SERVER.

If we would like to change the client we should modify the protocol. This is not very convenient because our protocols support switch interactions and we could take advantage of this feature in order to change, for instance, the clients that communicate with the server.

Using instead Template Global Types we can avoid this problem, defining the protocol as follows:

[3] The reader should ignore the 0 placed as second argument in the event type definition: the number is required to synchronize sequences present in different *fork* branches; in the examples in this paper no synchronization is necessary and the argument is fixed to 0.

```
SERVER =
(receive_request(var(1)),0):(serve_request(var(1)),0):SERVER.
```

In this way we have written a generic protocol where we can change the involved agents simply changing the domain of parameter $var(1)$.

The domain of $var(1)$ is set during the application stage. In fact, a Template Global Type must be "applied" in order to turn into a "normal" Constrained Global Type, which can be used as described in our previous work. In particular, after the application stage, the obtained Constrained Global Type can be projected. The projection of a Constrained Global Type is still a Constrained Global Type, where events not involving the agents in the given set are removed.

When we say "project on a set of agents", we mean that, being the protocol a static and global representation of our system, there are some parts of it that are interesting for some agents but not for others. In general, an agent is not present in all the protocol points and it is for this reason that, before allowing an agent A to "execute" a protocol, we project it onto A, in order to remove all events which do not involve A from the protocol instance that A will need to execute.

Let us consider a more complex example:

```
SERVER1 =
(receive_request(client1),0):(serve_request(client1),0):SERVER1,
SERVER2 =
(receive_request(client2),0):(serve_request(client2),0):SERVER2,
SERVER3 =
(receive_request(client3),0):(serve_request(client3),0):SERVER3,
SERVER = SERVER1|(SERVER2|SERVER3).
```

In this example, we have three protocol branches, each of which is similar to the simple protocol described before, combined using a "fork" operator. The first branch involves `client1`, the second `client2`, the third `client3`. As we can see we have to write the same piece of code many times and above all it is impossible to change agents at runtime, for example during a protocol switch.

Instead, using Template Global Type we may write:

```
SERVERT =
(receive_request(var(1)),0):(serve_request(var(1)),0):SERVERT,
SERVER = finite_composition(fork, SERVERT, [var(1)]).
```

The construct $finite\_composition$ is used to compose many times a Constrained Global Type with a chosen operator, in this case the fork operator. If we iterate the $var(1)$ parameter on the set containing $\{client1, client2, client3\}$ we get the same results as before, without $var(1)$. The great advantage of this approach, is that the set over which $var(1)$ ranges can be decided at runtime, hence allowing the agents to implement a (limited) form of **dynamic protocol generation**.

In order to better explain the application and projecting phases, we can see (with a short piece of *pseudocode*) some sample calls:

```
SERVERT =
        (receive_request(var(1)),0):
        (serve_request(var(1)),0):
        SERVERT,
SERVER = finite_composition(fork, SERVERT, [var(1)]),
apply(SERVER,
      [t(var(1), [client1, client2, client3])],
      INSTANTIATEDSERVER),
project(INSTANTIATEDSERVER, [agent1], PROJECTEDSERVER).
```

The *apply* predicate instantiates the `SERVER` protocol returning its instantiation in `INSTANTIATEDSERVER` variable. Afterwards, the obtained "normal" Constrained Global Type without parameters can be projected; in our example, the projection is on an agent called *agent1.*

After the application and projection phases we obtain a "customized" protocol driven agent. This agent will have to only choose what to do during its execution on the basis of what is expected by the protocol; the respect of the global protocol is guaranteed because each agent directly derives from it via projection, and each agent is guided by the same interpreter, that interrogates the Prolog library which implements both the protocol definition and the "apply", "project", and all the other predicates which are necessary in order to know what is expected from protocol and what is not.

The original contribution described in this paper lies in the design of Template Global Types and in the implementation of the apply predicate. The other predicates were already implemented. Also, we integrated the Template Global Types mechanism into both Jason and JADE.

4 Integration inside Jason

Before trying the integration in JADE, we considered a more linear and modular implementation in Jason.

Jason is an interpreter for an extended version of AgentSpeak, where each agent implements the BDI approach; it is a useful framework in order to quickly create MASs architectures using a Prolog-like language.

In case of Jason, the implementation was almost straightforward because it directly supports Prolog code. Hence, it was not necessary to add an interface between Jason and the Prolog engine, since the Prolog library defining apply, project, and all the other predicates necessary for implementing a protocol-driven behavior, could be used by Jason almost "as it is" (with definitely minor syntactic changes).

The integration of our approach into JADE was instead more challenging. We had many different design choices, but in the end we had to opt for the architecture in which most of the work is done in Prolog, and JADE is almost passive. The reason for this choice, better explained later, is that JADE cannot directly manipulate the representation of cyclic protocols, because of limitations of the Java-Prolog interface, hence we had to relegate all the operations on the protocols into the Prolog code, only task of calling Prolog predicates, without taking any decision.

5 Integration inside JADE

JADE is a Java framework where each agent must extend a common Java class (*Agent* class). We created a class called *AgentProtocolDriven* that extends it. Each new agent must extend the latter and override some methods:

- *setup*, method dedicated to initialize the Prolog engine with all predicates necessary to the agent;
- *react*, method dedicated to the agent reaction after a message reception that is expected by the protocol;
- *unexpected*, method dedicated to the agent reaction after a message reception that is unexpected by protocol;
- *select_messages*, method used by the agent in order to select which message to send between those expected by the protocol.

Each agent's setup method must recall the inherited method of its parent (*AgentProtocolDriven* class) in which the main behavior implementing the interpreter's body is created and added. This is a *Cyclic Behavior* which is executed any time the agent is selected by the JADE schedule. It is like a loop, and in each round the agent can check if can do something coherently with the protocol.

The parent's setup method does not only create a behavior but it cares about instantiation and projection of the protocol by calling the Prolog predicate: *instantiate_template_and_project*.

Below we show the Prolog code corresponding to this predicate. It can be easily seen that the code can be broken down into three basic components (as already seen in Figure 1):

- Protocol representation
- Protocol instantiation
- Protocol projection

```
/* Predicate that manage the instantiation
   of a Template Global Type */
instantiate_template_and_project
 (Name, ActualParameters, MyName, ProjectedAgents) :-
```

```
    /* Get the Template Global Type from the library,
       this protocol can have parameter variables */
    trace_expr_template(Name, GlobalType),
    /* Preprocessing phase where all the syntactic sugar
       and parameter variables are removed */
    apply(GlobalType, ActualParameters, InstantiatedTemplate),
    /* Project protocol on this agent */
    project(MyName, InstantiatedTemplate,
            ProjectedAgents, ProjectedGlobalType),
    /* Update current state of protocol */
    clean_and_record(MyName,
                     current_state(ProjectedGlobalType)).
```

The above code is invoked in Java as follows.

```
// Instantiate and project the protocol
new Query(
  "instantiate_template_and_project(" +
  protocolName + "," +
  protocolParameters + "," +
  getLocalName() + "," +
  "[" + getLocalName() + "])"
).hasSolution();
```

After this sequence of instructions, inside the Prolog engine the projected protocol of our agent is correctly instantiated and the interpreter can follow it.

To know the actions allowed by the protocol at a given time (namely, which messages the agent can send, which one it is allowed to receive), the agent queries the Prolog library where all the important pieces of information, like the current state of protocol, are maintained; to do this, in JADE, we have to use a Java library called JPL, that makes communication between a Java program and the SWI Prolog engine possible. So, the JADE agents interpreter can, step by step, ask to Prolog what the agent can or cannot do.

To receiving or sending a message is explicitly set a priori with other parameters, all by reading a configuration file.

5.1 Message reception

When a message is received by an agent, it is put in a queue. When a message is selected from the queue, in order to check if it is expected by the protocol in the current state we have created a predicate in Prolog that returns a list containing all messages that agent can receive.

```
/* Messages that agent can receive according to
   current state of the protocol */
inMsgs(MyName, ListToReceive) :-
 /* Current state of the agent protocol */
 recorded(MyName, current_state(LastState), _),
 /* Find all possible next state
```

```
    where agent is the receiver */
 findall(
  msg(SenderV, MyName, PerformativeV, ContentV, NewStateV),
  next(
   0, LastState,
   msg(SenderV, MyName, PerformativeV, ContentV), NewStateV,
   0, MyName),
  ListToReceive).

/* If Msg is allow in this state of protocol
   move to new state and save it */
move_to_next(MyName, Msg) :-
 /* Current state of the agent protocol */
 recorded(MyName, current_state(LastState), Ref),
 /* Try to do a step,
    if it is valid in the current state */
 next(0, LastState, Msg, NewState, 0, MyName),
 erase(Ref),
 /* Update current state of protocol */
 recorda(MyName, current_state(NewState)).
```

The JADE agent should only call *move_to_next* and execute the *react* method if move to next does not fail, and the *unexpected* method otherwise.

5.2 Messages sending

When an agent wants to send a message it must check which messages are allowed by the protocol in the current state. In order to do this, the Jade agent can call the *outMsgs* Prolog predicate that returns all messages that it can send in the current state of protocol.

```
/* Messages that agent can send according to
   current state of the protocol */
outMsgs(MyName, ListToSend) :-
  /* Current state of the agent protocol */
  recorded(MyName, current_state(LastState), _),
  /* Find all possible next state
     where agent is the sender */
  findall(
   msg(MyName, ReceiverV, PerformativeV, ContentV),
   next(
    0, LastState,
    msg(MyName, ReceiverV, PerformativeV, ContentV), _,
    0, MyName),
   ListToSend).
```

The Jade agent should only select one message from the list of messages returned by the predicate using the select messages method.

5.3 Problems encountered only with JADE

The interpreter implementation in JADE was more complicated than in Jason, indeed we found many more different problems.

The main problems can be summarized in two specific cases:

- the JPL library does not support cyclic terms;
- SWI Prolog assert predicate does not allow a cyclic term as argument.

It is easy to note that the second problem result from a lack of SWI Prolog in the management of cyclic terms.

In order to solve the first problem, we had to create "super predicates", which are simply collections of predicates, to ensure that all intermediate executions are made within Prolog and no cyclic term is returned to JADE.

The second problem was solved instead using another predicate inside SWI Prolog; indeed, the *record* predicate supports cyclic terms and has a behavior similar to the *assert* predicate.

6 Related Work

A large part of the state of the art analysis presented in this section was published in [11].

Our work falls in the research area on self-adaptive systems which spun off from the wider area of distributed systems, be them based on web services, software agents, robots, or on other autonomous entities that need to react to unforeseen changes during their execution. Many surveys have been conducted to identify the main features of self-adaptive MASs [12, 15, 22, 25, 26] and interesting and original solutions have been proposed by the research community.

Proposals for standardizing the concepts involved in the self-adaptation process include a meta-model to describe intelligent adaptive systems in open environments [16] and a taxonomy of adaptive agent-based collaboration patterns [7], for their analysis and exploitation in the area of autonomic service ensembles. An analysis of linguistic approaches for self-adaptive software is presented in [24].

The approaches closer to ours focus on formalizing protocols that the agents may use during their life, including specific protocols to deal with unforeseen events: in these approaches agents are usually free to choose, from a bunch of usable protocols, which one they prefer, maintaining in this way the freedom to autonomously self adapt to the new situation but ensuring at the same time that a feasible interaction pattern is followed. Our work can be included in this research field, where we can speak of "protocol enforcement" or "protocol-driven agents".

As far as self-adaptiveness of protocol-driven agents is concerned, the main sources of inspiration were [8, 9, 20, 21]. In [8] the authors propose a dynamic self-monitoring and self-regulating approach based on norms to express properties which allow agents to control their own behavior. In [20] and [21] agents operating in open and heterogeneous MASs dynamically select protocols, represented in FIPA AUML, in order to carry collaborative tasks out. Since the selection is performed locally to the agent, some errors may occur in the process. The proposed mechanism provides the means for detecting and overcoming them.

Comparison. To the best of our knowledge, there are no approaches similar to ours presented in the MAS literature.

In Fornara at al [13, 14, 19] the authors discuss their approach based on Normative MAS. An artificial institution catches the institutional events and verifies them with respect to a normative specification. As a result, protocol specifications are a special case with respect to a normative specification. So, even if the approach is different the aim is similar, i.e. to deal with open multiagent systems and monitor their correctness w.r.t. a specification.

Normative system approaches offer other advantages for multi agent systems because agents may integrate their practical reasoning with reasoning about the normative specification, although, also our protocol-driven agents could reason about trace expressions which are a First Class Entities.

Other closely related proposals are those by Criado et al. [10] and Baldoni et al. [4, 3, 17, 5]; in these papers, the authors suggest a way to implement a monitoring mechanism by exploiting the A&A metamodel and by reifying commitment-based protocols into artifacts. The proposal is implemented both on top of Cartago and Jade and on top of Jason/JaCaMo. However, our work is different from theirs, because – at least from the Runtime Verification application – our approach is less invasive, in fact, it works with each possible MASs architecture and not with only customized implementations.

If we consider our previous work, before upgrading with Template Global Types, one reason why our approach was different from others, was that the projection function took protocol specifications and returned protocol specifications expressed in the same language. Usually, projection functions return either agent stubs/code (common in the MAS community) or protocol specifications in a language suitable for expressing the agent local viewpoint, different from the language for expressing the global one (common in the session types community). Having a unique formalism for protocol specification both at the global and at the local level is a simpler and more uniform approach.

In this paper we have shown the benefits of using parameters inside protocol specifications; in this way we have made protocols much more generic and flexible, also moving a step towards dynamic protocol generation.

7 Conclusions

In this paper we have presented our proposal to make the management of protocols more flexible and to move a step forward their dynamic generation. Two working prototypes exist, demonstrating the feasibility of our approach. While integrating our parametric protocol-driven agents into Jason was easy because of its native support to Prolog, integrating them into JADE was not. However, that attempt – which, although not trivial, was successful – makes us confident in the possibility to integrate our approach into almost any agent framework, given that an interface between the framework language and Prolog is provided.

References

1. D. Ancona, M. Barbieri, and V. Mascardi. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In S. Y. Shin and J. C. Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1377–1379. ACM, 2013.
2. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring MASs from multiparty global session types in Jason. In M. Baldoni, L. A. Dennis, V. Mascardi, and W. Vasconcelos, editors, *Declarative Agent Languages and Technologies X - 10th International Workshop, DALT 2012, Valencia, Spain, June 4, 2012, Revised Selected Papers*, volume 7784 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 2012.
3. M. Baldoni, C. Baroglio, and F. Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Trans. Internet Technol.*, 14(4):23:1–23:23, Dec. 2014.
4. M. Baldoni, C. Baroglio, and F. Capuzzimati. Typing multi-agent systems via commitments. In F. Dalpiaz, J. Dix, and M. van Riemsdijk, editors, *Engineering Multi-Agent Systems*, volume 8758 of *Lecture Notes in Computer Science*, pages 388–405. Springer International Publishing, 2014.
5. M. Baldoni, C. Baroglio, F. Capuzzimati, and R. Micalizio. Programming with commitments and goals in JaCaMo+. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, pages 1705–1706, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems.
6. D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE and Jason multiagent systems with prolog. In L. Giordano, V. Gliozzi, and G. L. Pozzato, editors, *Proceedings of the 29th Italian Conference on Computational Logic, Torino, Italy, June 16-18, 2014.*, volume 1195 of *CEUR Workshop Proceedings*, pages 319–323. CEUR-WS.org, 2014.
7. G. Cabri, M. Puviani, and F. Zambonelli. Towards a taxonomy of adaptive agent-based collaboration patterns for autonomic service ensembles. In *Collaboration Technologies and Systems (CTS), 2011 International Conference on*, pages 508–515, 2011.
8. C. Chopinaud, A. El Fallah-Seghrouchni, and P. Taillibert. Automatic generation of self-controlled autonomous agents. In *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, pages 755–758, 2005.
9. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Self-adaptive monitors for multiparty sessions. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014*, pages 688–696. IEEE, 2014.
10. N. Criado, E. Argente, P. Noriega, and V. J. Botti. Reasoning about constitutive norms in BDI agents. *Logic Journal of the IGPL*, 22(1):66–93, 2014.

11. A. F. Davide Ancona, Daniela Briola and V. Mascardi. Global protocols as first class entities for self-adaptive agents. *AAMAS2015*, 2015.
12. G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-organization in multi-agent systems. *Knowl. Eng. Rev.*, 20(2):165–189, 2005.
13. N. Fornara, F. Vigan, and M. Colombetti. Agent communication and artificial institutions. *Autonomous Agents and Multi-Agent Systems*, 14(2):121–142, 2007.
14. N. Fornara, F. Vigan, M. Verdicchio, and M. Colombetti. Artificial institutions: a model of institutional reality for open multiagent systems. *Artificial Intelligence and Law*, 16(1):89–105, 2008.
15. M.-P. Gleizes. Self-adaptive complex systems. In M. Cossentino, M. Kaisers, K. Tuyls, and G. Weiss, editors, *Multi-Agent Systems*, volume 7541 of *Lecture Notes in Computer Science*, pages 114–128. Springer Berlin Heidelberg, 2012.
16. T. Juan and L. Sterling. The ROADMAP meta-model for intelligent adaptive multi-agent systems in open environments. In P. Giorgini, J. Müller, and J. Odell, editors, *Agent-Oriented Software Engineering IV*, volume 2935 of *Lecture Notes in Computer Science*, pages 53–68. Springer Berlin Heidelberg, 2004.
17. C. B. M. Baldoni and F. Capuzzimati. Reasoning about social relationships with Jason. *Autonomous Agents and Multi-Agent Systems*, 2014.
18. 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.
19. D. Okouya, N. Fornara, and M. Colombetti. An infrastructure for the design and development of open interaction systems. In M. Cossentino, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Engineering Multi-Agent Systems*, volume 8245 of *Lecture Notes in Computer Science*, pages 215–234. Springer Berlin Heidelberg, 2013.
20. J. G. Quenum, S. Aknine, O. Shehory, and S. Honiden. Dynamic protocol selection in open and heterogeneous systems. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Hong Kong, China, 18-22 December 2006*, pages 333–341, 2006.
21. J. G. Quenum, F. Ishikawa, and S. Honiden. Protocol selection alongside service selection and composition. In *2007 IEEE International Conference on Web Services (ICWS 2007), July 9-13, 2007, Salt Lake City, Utah, USA*, pages 719–726, 2007.
22. R. de Lemos, H. Giese, H. A. Müller, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32, 2013.
23. A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In V. R. Lesser and L. Gasser, editors, *Proceedings of the First International Conference on Multiagent Systems, June 12-14, 1995, San Francisco, California, USA*, pages 312–319. The MIT Press, 1995.
24. G. Salvaneschi, C. Ghezzi, and M. Pradella. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.*, 8(2):7:1–7:29, 2013.
25. D. Weyns and M. Georgeff. Self-adaptation using multiagent systems. *Software, IEEE*, 27(1):86–91, 2010.
26. F. Zambonelli, N. Bicocchi, G. Cabri, L. Leonardi, and M. Puviani. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2011 Fifth IEEE Conference on*, pages 108–113, 2011.