Global Protocols as First Class Entities for Self-Adaptive Agents

Davide Ancona Daniela Briola Angelo Ferrando Viviana Mascardi
DIBRIS, University of Genova
Via Dodecaneso 35, 16146, Genova, Italy
davide.ancona@unige.it,daniela.briola@unige.it,
angelo.ferrando42@gmail.com,viviana.mascardi@unige.it

ABSTRACT

We describe a framework for top-down centralized self-adaptive MASs where adaptive agents are "protocol-driven" and adaptation consists in runtime protocol switch.

Protocol specifications take a global, rather than a local, perspective and each agent, before starting to follow a new (global) protocol, projects it for obtaining a local version. If all the agents in the MAS are driven by the same global protocol, the compliance of the MAS execution to the protocol is obtained by construction.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems

General Terms

Design; Languages; Reliability

Keywords

Self-adaptiveness; Interaction Protocols

1. INTRODUCTION

Today's software systems raise many challenges to their designers as they are required to be more and more autonomous, recoverable and reliable to guarantee the expected level of the offered services. Achieving all the three goals together requires to find the right balance between the ability of the system to operate with a high degree of freedom, including its ability to recover to an acceptable state in case of exceptional situations, and the guarantee of a behavior compliant with the designers' requirements.

Self-adaptive systems, namely systems able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals [40], are a widely accepted answer to the increasing need of autonomy and recoverability of modern complex systems.

Reliability can be achieved by enforcing all the system's components to respect given patterns of behavior, known to be safe.

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May 4–8, 2015, Istanbul, Turkey.*

In this paper we address the design and implementation of self-adaptive multiagent systems where compliance to a given interaction protocol is guaranteed by construction, allowing agents to operate in a reliable way. In the proposed framework, agents are driven by first class specifications of interaction protocols which can change at runtime, making self-adaptation a side result obtained almost for free.

Adaptation takes place according to the instructions of agents empowered to request protocol switches and acting as central controllers. In the most general setting, different agents could take the role of controller in different moments, or even in the same moment provided that they coordinate themselves for controlling the system. In our framework we assume that there is only one controller agent at a time. Since switching to a new protocol may require to be in some safe state[1], the protocol's designer can specify when the agents are allowed to manage a switch request (and, as a consequence, when they are not).

The code of "protocol-driven" agents is not generated from the protocol specification prior to the MAS deployment, as this would hard wire the protocol-compliant behavior into the code preventing agents from adapting to protocol changes. Rather, the agent's interpreter takes the current state of the interaction protocol into account to devise which messages could be sent, allowing the agent to select and send one of them, or which messages could be received, allowing the agent to verify whether the received message – if any – was one among the expected ones and to react in a suitable way. Changing the executing protocol as the result of an acceptable protocol switch request causes the interpreter to call a cleanup procedure and to proceed in the normal way following the new protocol.

A characterizing feature of our approach is that protocol specifications take a global, rather than a local, perspective and each agent, before starting to follow a new protocol, projects the protocol onto itself by removing the protocol's components not involving itself. If all the agents in the MAS are driven by the same global protocol, the compliance of the MAS execution to the protocol comes for free.

The paper is organized as follows: Section 2 discusses the related literature. Section 3 provides a gentle introduction to our framework. Section 4 provides technical details on

[1]Consider for example an agent following the protocol "drive home" who is required by his kid in the backseat to move to the protocol "look at me, I feel bad...". Even if looking at the suffering kid is a high priority protocol to follow, the agent can move to it only when the car's speed is low enough, or, better, the car has been stopped.

protocol specifications and Section 5 discusses self-adaptive protocol-driven agents. In Section 6 the proposed approach is explained by means of an example and its feasibility is demonstrated by the implementation in Jason. Section 7 concludes.

2. RELATED WORK

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 [21, 24, 40, 45, 48] and interesting and original solutions have been proposed by the research community.

Proposals for standardizing the concepts involved in the self-adaptation process include [29], a meta-model to describe intelligent adaptive systems in open environments, and [10], a taxonomy of adaptive agent-based collaboration patterns, for their analysis and exploitation in the area of autonomic service ensembles. An analysis of linguistic approaches for self-adaptive software is presented in [41].

As far as the engineering of self-adaptive systems is concerned, authors either propose new methods and platforms or extend already existing methodologies. In the first category we mention the AMAS theory (Adaptive Multi-Agent Systems [11]), which identifies design criteria to enable the emergence of an organization within the system and to guarantee the global function of the system even in critical situations, Unity [43], a decentralized architecture for autonomic computing based on multiple interacting agents, and DSOL [18], a declarative approach supporting dynamic service orchestration at run-time. As good representatives of the second category we may mention [14], based on Gaia [49], and [19, 36], based on Tropos [7].

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". There are at least three ways to obtain a protocol-driven behavior:

1. automatically generating the agent's code from the protocol specification in order to ensure the compliance by construction;

2. monitoring the agents interactions to identify a violation and enforcing a recovery;

3. making agents capable of directly executing or interpreting the formal protocol specification.

Proposals following the first approach received the attention from the community since the Agent-Oriented Software Engineering early days. The PASSI methodology [17] allows the designer to generate the agent's structure and its internal code starting from UML models and a similar functionality is offered by Prometheus [37]; West2East [12] offers libraries for both the translation of protocols represented in FIPA AUML[2] into different textual notations, and the automatic generation of an executable program compliant to the original interaction protocol; Dsml4Mas [44] can be used to design protocols following a model-driven approach for generating executable code from protocol specifications [26].

The second approach is discussed for example in [9], where a mechanisms for monitoring norms is proposed, in [25], where the global-level adaptation is based on the monitoring of the system's behavior and on a dynamic reification of an organizational structure, and in [20], where a form of agent supervision, which constrains the actions of an agent so as to enforce certain desired behavioral specifications, is presented.

We followed the third approach, namely allowing agents to directly executing or interpreting the protocol. The related literature we are aware of is all centered around protocols conceived as first class entities and represented by means of commitments. For example in [4] commitment protocols can be directly accessed by the agents, as they are artifacts available in the platform, whereas in [47] agents use a "commitments plus axioms" protocol representation that enables them to flexibly accommodate the exceptions and opportunities that may arise at run time. A framework for modeling and handling exceptions is presented in [30] and in successive works by Singh et al. [23, 42], where a formal methodology is proposed to manage, even at run time, expected and unexpected exceptions in commitment-based MASs. Although these approaches are close to ours in spirit, commitment protocols are definitely different from constrained global types, which are inspired by global and multiparty session types [13] and include no notion of commitment. Also, extending existing agent environments or languages by implementing our framework on top of them should be always possible as long as these environments meet the few – and almost mandatory for any MAS – conditions stated in Section 3. Implementing an approach based on commitment protocols, instead, requires a paradigm shift, where agents are designed and implemented adhering to the commitment approach.

In [33, 34, 35] interaction protocols are modeled as executable entities that can be referenced, inspected, composed, shared, and invoked between agents. The $\mathcal{RASA}$ formalism defined in those papers is close to ours, even if its expressiveness is lower as no fork operator is supported. Also, no exploitation of the framework on top of real agent systems is presented.

As far as self-adaptiveness of protocol-driven agents is concerned, the main sources of inspiration were [15, 16, 38, 39]. In [15] 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 [38] and [39] 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.

Our work is similar to [16] both in the formalism used to represent protocols and in the idea of dynamic protocol switch, but our solution is actually implemented and thus must take many more practical aspects - for example what to do before switching to a new protocol, how to state when the agent is in a safe state and can actually perform a protocol switch - into account.

[2] `www.auml.org/auml/documents/ID-03-07-02.pdf`.

Comparison. To the best of our understanding, none of the mentioned approaches covers the three stages of starting from a global description of the protocol, moving to local versions for the individual agents via projection, and interpreting these local versions on an actual agent framework to drive the agents' behavior, as we do. One reason why our approach is different from others, is that the projection function takes protocol specifications and returns 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 the MAS area, when the code of an agent is generated from a protocol specification, the language used for specifying the interaction protocol and the language used for implementing the agents are, again, different. The agent is "compiled" into some AOP language starting from the protocol's specification. On the contrary we do not generate any agent code into any AOP language. The protocol specification is interpreted and this gives the flexibility that meta-programming ensures, demonstrated for example by the easiness in implementing protocol switch.

Finally, w.r.t. our own previous work, in [1] and [3] (resp. [2]) we proposed to exploit the interaction protocol formalism (resp. the protocol projection mechanism) as a way to face (resp. to make more efficient) the runtime verification of protocol compliance via monitoring. This paper, instead, is not concerned with runtime verification and monitoring. Also, issues like self-adaptiveness, protocol driven agents, protocols as first class entities and protocol switch are not addressed by [1, 2, 3]. This paper and the previous ones share the adoption of the same formalism (constrained global types) and tools (projection and `next` functions), but used for definitely different purposes.

3. OUR FRAMEWORK

Our framework addresses top-down self-adaptive systems. A system of this kind "*is often centralized and operates with the guidance of a central controller or policy, assesses its own behavior in the current surroundings, and adapts itself if the monitoring and analysis warrants it. Such a system often operates with an explicit internal representation of itself and its global goals*" [46].

In our framework agents able to adapt are "protocol-driven": they are characterized by one interaction protocol specified in some suitable formalism and by three mandatory components, the *knowledge base*, the *message queue* and the *environment's representation*, that should be directly implemented in the underlying agent framework. As we make no other assumption on the agent's architecture, our framework is as general as possible and could be implemented in any underlying environment or programming language where these three components are available (namely almost all the agents' frameworks, be them BDI-oriented like Jason [6], or not BDI-oriented like JADE [5]).

Besides protocol-driven agents, also "normal" agents entirely implemented in the underlying framework can be part of the self-adaptive MAS. For example, since our protocol-driven agents are implemented on top of Jason, we could create a MAS were normal Jason agents defined in a standard way using mental notes, plans, goals, co-exist with protocol-driven agents which, albeit being implemented as Jason agents as well, must follow a specific syntax and behavior (see Section 6). Normal agents give the possibility to reuse existing code and to implement behaviors that cannot be conveniently modeled using an interaction protocol, for example because they must access legacy code or perform complex computations. However, no hypotheses can be made on their adaptability and reliability.

Being protocol-driven means that the agent behaves according to a given protocol. In each time instant, the protocol-driven agent can make only those internal choices which are allowed by the protocol in the current state. In case of events which depend on external choices, the agent can only verify if the event that took place is compliant with the protocol and act consequently. Events could be in principle of any kind but in order to demonstrate the feasibility of our approach in a neat way, in this paper we limit ourselves to consider interaction events, where sending events are the result of an internal choice, whereas reception events are not under the agents' control and can be neither prevented nor forced, but only checked.

Adaptation takes place when the protocol-driven agent switches to a new protocol on request of the controller agent, which might follow a MAPE-like loop [27, 46]. In this paper we make no assumptions on the controller's internal architecture and functioning. For our purposes, the controller is just an agent which has the power to request protocol switches to some protocol-driven agents and which may or may be not protocol-driven, depending on which type of behavior it implements and which requirements it must meet.

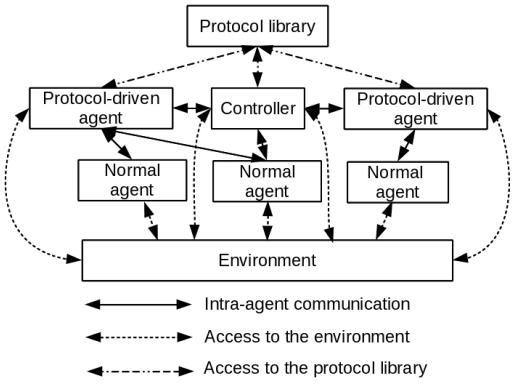

Figure 1: Architecture of a top-down, centralized self-adaptive MAS.

The components of our framework are shown in Figure 1. The protocol library may be either external, like in the figure, or hard-wired in the controller's knowledge base. From a logical viewpoint this makes little difference. In the first case, when the controller identifies which protocol the agents must follow to adapt to a new situation, it communicates the protocol's identifier to the agents which will retrieve its representation from the library. In the second case, it will send the full protocol's representation inside the message's content.

Protocols in the library always take a global perspective (namely, a perspective where all the parties involved in the protocol are managed in a homogeneous way, without tak-

ing the point of view of one of them), but may involve a subset of the agents in the MAS: the controller may send different protocols to different subsets of agents, even if this requires a careful design of such protocols to avoid unwanted interferences.

Protocol-driven agents interact with all the agents, including the controller and normal ones, via message passing. This requires that the controller and normal agents are modeled in the protocol, making protocol-driven agents aware of them.

Being protocol-driven does not contrast with the main agents' features identified by N. R. Jennings, K. P. Sycara and M. Wooldridge in [28]:

– *Situatedness* is achieved by setting the agent's policies defining how to select a message to send among the possible ones and what to do when a message allowed by the protocol is received. These policies take information coming from the environment into account and can affect it as a side effect.

– *Autonomy* is preserved because protocols usually define different allowed patterns of interactions, without imposing the choice of which of them following: the choice is left to the agent, thus balancing its respect of the protocol and its autonomy.

– *Responsiveness* and *proactiveness* depend on the protocol itself. For example, a protocol describing the interactions between Alice and Bob, where Alice always sends a message a to Bob and Bob always receives it and does nothing, leaves room neither for responsiveness in Alice behavior, nor for proactiveness in Bob's one. It is up to the protocol designer to cope with these issues in a proper way.

– *Sociality*, namely the ability to interact, when appropriate, with other agents and humans, is of course the main requirement for conceiving communication-intensive agents like protocol-driven ones and it is the assumption under which our proposal works.

For supporting a protocol-driven approach to agent programming, a formalism for expressing protocols must exist together with a *generate* function for identifying the allowed actions (both sending and receiving) for moving from the current state of the protocol to the next one. What differentiates the behavior of each agent are the *select* policy to select the message to send among the allowed ones, and the *react* policy to react to an incoming message. Two more policies must be defined to state how to manage *unexpected* messages and which *cleanup* actions to perform before switching from the currently executing protocol to the new one.

We assume that each agent Ag in the MAS is able to project a global description of a protocol involving many agents onto a local version by keeping only interactions that involve Ag. Like the *generate* function, also the *project* one can be provided by some artifact in the MAS or may be implemented by the agent itself.

If a description of the global protocol that all the agents in the MAS must respect exists, the local protocol for each agent can be automatically obtained from the global one. This allows the whole MAS to respect the global protocol by construction.

The architecture of a protocol-driven agent is depicted in Figure 2. The interpreter implements a cycle where it first checks if there is a protocol switch request and if it can be managed in the current state of the protocol. If yes, and if the sender has the power to make such a request - namely, if it is a system's controller -, a protocol switch is performed after some cleanup operations. If no protocol switch is foreseen by the protocol in that moment, or no protocol switch request has been received, the normal communication actions that can be performed are generated and, according to the precedence that the agent gives to receiving or sending (in case both options are available), one of them is performed. The environment representation and knowledge base are updated accordingly and the protocol moves to the next state. In case the received message was not foreseen by the protocol, it is managed according to the unexpected policy. An alarm which is reset any time an action is performed allows to avoid deadlocks, for example in case the agent gives the precedence to receiving, but no messages are available in the message queue. When the alarm expires the agent cannot wait any longer and selects one of the possible messages to be sent (if any), forcing itself to adopt a sending precedence.

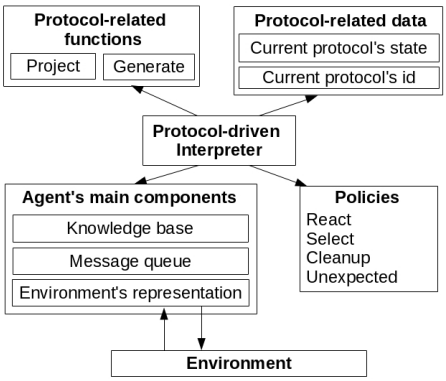

Figure 2: Architecture of a protocol-driven agent.

In order to move from a general description of the framework to its implementation, we need to fix some choices, in particular the protocol representation formalism (and, as a direct consequence, the *generate* and *project* functions) and the underlying agent framework. Our implementation instantiates the general one in the following way:

– All the protocol-driven agents use the same protocol formalism and are able to perform generation and projection by themselves.

– There is an external protocol library.

– The formalism for expressing protocols is that of "constrained global types" introduced in [1, 3] and the *generate* function is a variant of the `next` transition function defined in those papers.

– The projection algorithm is the one defined in [2]. Projection can be described as a function $\Pi : \mathcal{T} \times \mathcal{P}(AGS) \rightarrow \mathcal{T}$ where $\mathcal{T}$ is the set of constrained global types. Given a constrained global type τ and a set of agents Ags as input, Π returns a constrained global type τ_{Ags} which contains only interactions involving agents in Ags: interactions that do not involve agents in Ags are removed from τ_{Ags}.

– The underlying agent's framework where our self-adaptation approach has been experimented is Jason.

4. PROTOCOL SPECIFICATIONS

Protocols are formalized using constrained global types. States of the protocol are represented by constrained global types as well, blurring the distinction between "protocol" and "protocol state". For requesting a protocol switch we

introduce a special `switch` performative with a protocol as content. Like any other interaction, protocol switches may be allowed in some points of the protocol and not in others, making it possible for the agents to switch to a new protocol only at the right time.

A protocol is characterized by its identifier, its specification and the definition of interaction types identifying interactions allowed in a specific state of the protocol.

Interactions have the form `msg(Sender, Receiver, Perf, Content)`. `Sender` and `Receiver` are agent identifiers, `Perf` is a performative in the agent communication language supported by the underlying framework (for example, KQML [32] in Jason and FIPA-ACL [22] in JADE), `Content` is the interaction content in some suitable representation language. This model supports point to point communication only, although a broadcast operator could be defined on top of that.

We use the term "interaction" rather than "message" to stress that a protocol always represents a global description of what is expected to go on: the notion of interaction summarizes that one agent sends a message and another is expected to receive it. From the viewpoint of one single agent *Ag*, an interaction corresponds to an incoming message if *Ag* is the receiver, and corresponds to a message that *Ag* is expected to send, if it is the sender.

For requesting a protocol switch, `msg(Sender,Receiver, switch,Protocol)` is used: `Sender`, which must have the power to request a protocol switch to `Receiver`, wants that `Receiver` starts behaving according to `Protocol` as soon as possible (we will explain what "as soon as possible" means in Section 5).

Interaction types add an abstraction level between the actual interactions taking place in the MAS and the protocol specification. For example, `ask_enter_treasure(hobbit1)` is the interaction type of `msg(hobbit1,bilbo,ask,enter_treasure)`. We state that `msg(hobbit1, bilbo, ask, enter_treasure)` ∈ `ask_enter_treasure(hobbit1)` and we use the latter in the protocol specification.

If more than one hobbit, say `hobbit1`, `hobbit2` and `hobbit3`, could ask Bilbo to enter the treasure room and the protocol did not need to distinguish among them, we could introduce an interaction type `ask_enter_treasure` not depending on the hobbit, and state that the three interactions `msg(hobbit1,bilbo,ask,enter_treasure)`, `msg(hobbit2, bilbo,ask,enter_treasure)` and `msg(hobbit3,bilbo,ask, enter_treasure)` have interaction type `ask_enter_treasure`. When the protocol is in a state where `ask_enter_treasure` is allowed, any actual interaction `msg(hobbit`$_i$`,bilbo,ask, enter_treasure)`, with `i` ∈ {1, 2, 3}, can take place.

The protocol specification *PrSpec* represents a set of possibly infinite traces of interactions and is defined on top of the following operators:

- λ (*empty trace*), representing the singleton set $\{\epsilon\}$ containing the empty trace ϵ of interactions.
- $IntType^n$*:PrSpec* (*sequence with producer*), representing the set of all traces whose first interaction *Intr* matches the interaction type *IntType* (*Intr* ∈ *IntType*), and the remaining part is a trace in the set represented by *PrSpec*. The integer n specifies the least required number of times *Intr* ∈ *IntType* has to be "consumed" to allow a transition labeled by *Intr*. 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 between branches of the fork operator, *without introducing new interactions*.
- *IntType:PrSpec* (*sequence with consumer*), representing the set of all traces where *Intr* ∈ *IntType*, and the remaining part is a trace in the set represented by *PrSpec*. *IntType* must match with a producer $IntType^n$ interaction type available in another fork branch of the protocol.
- $PrSpec_1 + PrSpec_2$ (*choice*), representing the union of the traces of $PrSpec_1$ and $PrSpec_2$.
- $PrSpec_1 | PrSpec_2$ (*fork*), representing the set obtained by shuffling the traces in $PrSpec_1$ with the traces in $PrSpec_2$.
- $PrSpec_1 \cdot PrSpec_2$ (*concat*), representing the set of traces obtained by concatenating the traces of $PrSpec_1$ with those of $PrSpec_2$.

Section 6 provides examples of protocols represented using this formalism.

The focus of [1, 3] was on testing whether the actual interactions in the MAS were compliant with the protocol specification. To define the operational semantics of the protocol, and to implement the monitoring activity, a *next* transition function was defined as:

next: $Pr \times Intr \rightarrow Pr$

Next takes a protocol and the sniffed interaction *Intr*, and returns a new protocol where only the *PrSpec* component changed to represent the new state (if any) where the protocol can move given that *Intr* took place. To make some examples,

$next(\alpha^0{:}Pr, a) = Pr$ if $a \in \alpha$

$next(Pr_1 + Pr_2, a) = Pr$ if $next(Pr_1, a) = Pr$ or $next(Pr_2, a) = Pr$

In protocol-driven agents, the purpose of the protocol is no longer to test the acceptability of an interaction which already took place, but rather to generate all the interactions which are allowed in the current protocol state in order to drive the agent's behavior. This goal can be easily achieved by generating all the interactions which are allowed solutions of *next*, given the current state of the protocol. We define a *generate* function

generate: $Pr \rightarrow \mathcal{P}(Intr \times Pr)$

which takes one protocol Pr and returns the set of couples of interactions and protocols, $\{ (Intr_1,Pr_1), (Intr_2,Pr_2), ..., (Intr_n,Pr_n) \}$ such that $next(Pr, Intr_i) = Pr_i$.

Given the agent *Ag* who calls *generate*, we indicate with *InMsgs* the interactions returned by *generate* where *Ag* is the receiver, and with *OutMsgs* the interactions returned by *generate* where *Ag* is the sender. Intuitively, *InMsgs* are the messages that *Ag* expects in that protocol's state (it cannot decide which one among them to receive, but it can verify that the received message is one of them), and *OutMsgs* are the messages that *Ag* can decide to send. All the interactions returned by *generate* are associated with the state where the protocol would move if that interaction took place, in order to properly update the protocol's state.

5. SELF-ADAPTIVE PROTOCOL-DRIVEN AGENTS

A protocol-driven agent *Ag* is characterized by the following components:

- A unique agent identifier.
- A knowledge base *KBase* supporting the operations $?K$ (is knowledge formalized as K available in *KBase*, and which actual value is associated with it?), $+K$ (add knowledge formalized as K to *KBase*) and $-K$ (remove knowledge for-

malized as K from *KBase*). We assume that the knowledge base includes information on

1. the precedence policy stating which kind of communicative action (sending, `prec(send)`, or receiving, `prec(rec)`) the agent should give the precedence to, in case the protocol's state allows interactions of both kinds. Like any other piece of knowledge in the knowledge base, this information may change during time. However, some protocols[3] work only when the agents involved in them follow consistent precedence policies, and changing the policy at runtime may lead to deadlocks or unwanted behaviors;

2. the timeout `timeout(T)` stating for how long the agent can wait for being able to either sending or receiving a message; the agent's alarm is set to `T` any time a communicative action takes place and expires after `T` time units;

3. the list of agents who, in Ag's opinion, have the power to impose a protocol switch (`empowered(AgList)`). This list can change at runtime, for example because the trust of Ag towards some agents changes.

• A representation *Env* of the environment supporting external actions that the agent performs on the environment via effectors, and sensing actions performed by the agent via sensors. As customary, we make no assumptions on how sensors and effectors are implemented and we leave out from our investigation how the environment representation is kept consistent with the actual environment's state.

• The currently executing protocol Pr, whose syntax has been introduced in Section 4.

• A message queue *MsgQueue* storing interactions *Intr* corresponding to incoming messages that still have to be processed. The syntax of *Intr* is given in Section 4. Switch protocol requests have highest priority and always reach the top of the queue, when pushed into it. These messages are the only ones which are not tagged as "unexpected" when received in a protocol's state where they were not expected. In this case, they are locally stored[4] and managed as soon as the protocol reaches a state where a switch protocol request can be accepted.

• A received message reaction policy, stating how to update the knowledge base and the environment's representation as a consequence of the reception of one incoming message among the allowed ones (as returned by the *generate* function). The policy implementation may require to perform internal or sensing actions and, as a side effect, may affect the agent's knowledge base and the environment. No communicative actions can be performed as part of the policy.

react: $Intr \times KBase \times Env \rightarrow KBase \times Env$

• An outgoing message selection policy, stating which outgoing interaction (if any) among the allowed ones the agent will perform. The selection policy depends on the knowledge base and on the environment's representation and affects both of them. Like for the reaction policy, its implementation may rely on internal and sensing actions but not on communicative ones. The selection policy returns `null` in case either there are messages allowed by the protocol but none of them can be sent for reasons wired into the agent's selection function, or there are no messages at all.

select: $\mathcal{P}(Intr \times Pr) \times KBase \times Env \rightarrow$
$((Intr \times Pr) \bigcup \{\text{null}\}) \times KBase \times Env$

• A cleanup policy stating what actions the agent should perform before switching to another protocol.

cleanup: $KBase \times Env \rightarrow KBase \times Env$

• An unexpected message management policy stating what to do when a normal message not foreseen by the protocol is received, or when a switch message is sent by an agent who has not the power to act as a controller. The policy may vary according to the protocol currently under execution.

unexpected: $Intr \times KBase \times Env \rightarrow KBase \times Env$

The operational semantics of protocol-driven agents is given by the following interpreter which can call the *generate* and *project* functions.

For clarity of the presentation, we leave the knowledge base and environment components out of the arguments and return values of *react*, *select*, *cleanup*, *unexpected*. All these functions take the current knowledge base and environment and can update them.

Let KBase_0, Env_0, Pr_0, MsgQueue_0 be the initial knowledge base, environment, global protocol and message queue of Ag, respectively. A SwitchMsg variable is used to store the pending protocol switch request. An alarm (the Al variable) is initially associated with the Timeout value set by the agent's designer. A mechanism for checking whether the Al expired should be available.

Following Prolog's syntax, we use the "_" symbol to identify variables whose value does not matter.

Initialization

KB = KBase_0; En = Env_0; Pr = project(Pr_0, {Ag}); MQ = MsgQueue_0; SwitchMsg = null; ?timeout(T); Al = set(T);

Interpreter

```
while true {
/* C1: If a switch protocol request has been received, it is assigned to SwitchMsg and removed from the message queue */
  if (top(MQ) == mgs(S, Ag, switch, PrSwitch))
        SwitchMsg = pop(MQ);

/* The couples (interaction, next prot. state) allowed in the current state of the protocol are generated. For sake of presentation, we divide them into those involving incoming messages (InMsgs) and those involving messages that can be sent (OutMsgs) */
  InMsgs ⋃ OutMsgs = generate(Pr);

/* C2: If the reception of a switch protocol request is allowed in the current state of the protocol, and there is such a request, it is managed: SwitchMsg is reset */
  if (SwitchMsg == mgs(S, Ag, switch, PrSwitch) ∧
      (mgs(S, Ag, switch, PrSwitch), _) ∈ InMsgs)
       { SwitchMsg = null;
          /* C2.1 If the sender has the power to request a protocol switch, then the cleanup actions are performed according to the "cleanup" policy, the protocol is changed to the projection of PrSwitch onto Ag, and the current interpreter cycle is exited, otherwise the protocol switch request is managed by the "unexpected" policy */
          if (?empowered(AgList) ∧ S ∈ AgList)
```

[3]For example the protocol where Alice sends an arbitrary number of "ping" to Bob who answers with the same number of "pong", and then the protocol starts again, requires a synchronization between Alice and Bob in such a way that Bob avoids answering as soon as it receives a "ping" message, because Alice might want to send more "ping"s: Alice should give the precedence to sending and Bob to receiving.

[4]We assume that agents cannot have more than one pending protocol switch request at a time. Although the assumption is strong, it allows us to keep the algorithm simple. It can be relaxed using a queue of pending requests instead of a variable.

/* Protocol switch */
{ cleanup(); Pr = project(PrSwitch, {Ag}); }
else unexpected(mgs(S, Ag, switch, PrSwitch));
continue;
}
/ Note that, if condition C2 is not met, namely either there is no switch request or the protocol does not allow to manage it, nothing is done. If there is a switch request, it remains associated with variable SwitchMsg for being managed later, when the protocol will allow it */*

/ C3: If the message queue is empty, the alarm has expired, and there are no messages to send, the agent is stuck: the interpreter exits its main loop */*
if (empty(MQ) ∧ expired(Al) ∧ select(OutMsgs) == null;)
break;

/ C4: If the message queue is empty, the alarm has not expired, and the agent gives the precedence to reception, the current interpreter cycle is exited in the hope that, at the next one, some message will be available in the message queue; there is no risk to always remain in this condition as the alarm sooner or later will expire */*
if (empty(MQ) ∧ ¬expired(Al) ∧ ?prec(rec))
continue;

/ C5: If the message queue is not empty and the top message is not among those expected by the protocol (condition (top(MQ),_) ∉ InMsgs which includes the case InMsgs == ∅), the message is unexpected. It is popped and managed according to the agent's "unexpected" policy */*
if (¬empty(MQ) ∧ (top(MQ),_) ∉ InMsgs)
{ Msg = pop(MQ); unexpected(Msg); continue; }

/ C6: If the message queue is not empty, the top message is among the expected ones (this condition is superfluous but allows us to name the Pr_n component for successive use) and either the agent gives the precedence to reception and the current state of the protocol allows to receive messages, or the agent gives the precedence to sending but there are no messages to send, the first message in the queue is popped, the knowledge base and environment are updated according to the "react" policy, the protocol moves to the new state Pr_n and the alarm is reset */*
if (¬empty(MQ) ∧ (top(MQ),$\mathrm{Pr_n}$) ∈ InMsgs ∧
(?prec(rec) ∨ (?prec(send) ∧ select(OutMsgs) == null))
{ Msg = pop(MQ); react(Msg); Pr = $\mathrm{Pr_n}$;
?timeout(T); Al = set(T); continue; }

/ C7: The interpreter reaches this point if either the agent gives the precedence to sending, or it cannot wait for receiving messages because the message queue is empty and the timeout expired: the outgoing message selection is performed according to the "select" policy; the result of the selection cannot be* `null` *otherwise C3 would have been verified; the selected message is sent; the protocol moves to the next state; the alarm is reset */*
(Msg,$\mathrm{Pr_n}$) = select(OutMsgs); send(Msg);
Pr = $\mathrm{Pr_n}$; ?timeout(T); Al = set(T); }

6. IMPLEMENTATION

The implementation of our framework in Jason is available at `www.disi.unige.it/person/MascardiV/Software/selfAdaptiveAgents.html`. The `agent_protocol_interpreter.asl` file, which each protocol-driven agent in the MAS must include, contains the Jason code implementing the interpreter defined in Section 5 together with the *project* and *generate* functions. To make the prototype simpler, the specifications of all the available protocols are defined in a `protocol_library.asl` file, included by each agent which has then direct access to the protocols definitions.

Each agent must provide the definitions of the *react*, *select*, *cleanup*, *unexpected* policies and the beliefs related to precedence between sending and receiving, timeout, empowered agents. Given `P` the name of the initial protocol, the agent behavior is determined by the following piece of code (one initial goal and one plan, using Jason's terminology):

```
!start.
+!start : true <- !execute(protocol(P)).
```

Calling `!execute` starts the protocol-driven interpreter defined in `agent_protocol_interpreter.asl`.

In order to show the potential of our approach, we implemented an example where agents are freely inspired by characters from the Lord of the Rings. The purpose of this toy example is to explain how our approach works in practice, in a simplified and fictitious scenario. Extensions of constrained global types have been used to model real MASs in real domains [31]: more complex problems in more challenging settings like control access policies, e-commerce, ambient intelligence, resource allocation, can be addressed using constrained global types for protocol-driven agents as well.

The notation used for specifying protocols is the one introduced in Section 4. We describe the normal protocol **bh** involving Bilbo and the hobbits, whereas, for space constraints, we do not specify Frodo's, Folco's and Sam's normal behavior that we assume to be governed by **fr**, **fo**, **sa** protocols. The exceptional protocol **ef** involves Bilbo, Frodo, Folco and Sam. Gandalf acts as the controller of the MAS and may require a protocol switch from the normal protocol to the exceptional one.

Protocol involving Bilbo and the hobbits. To enter the treasure room, any hobbit must ask Bilbo and must respect his decision to either let him in or not. In the first case the hobbit thanks Bilbo and visits the room; in the second case he expresses his disappointment and may either start the protocol again, or give up. Bilbo allows in only one hobbit per day so the only knowledge he needs to check and update is related to whether one hobbit already entered the room today or not. Of course Bilbo can adopt other policies for deciding when and why allowing the hobbits in. We made experiments with different ones.

The branch of the protocol specifying the interactions between Bilbo and Hobbit1 is modeled by the following global type with only producer interaction types of the form $IntType^0$, written as `IntType^0` in the code excerpt, meaning that they require no consumer (see Section 4) to synchronize with.

```
Hobbit1Branch =
ask_enter_treasure(hobbit1)^0:
 ((ok_enter(hobbit1)^0:thanks(hobbit1)^0:lambda) +
  (no_enter(hobbit1)^0:grunt(hobbit1)^0:
     (lambda+hobbit1Branch)))
```

The branches for all the other hobbits are the same as `Hobbit1Branch` apart from the presence of `hobbit2`, `hobbit3`, ..., instead of `hobbit1`.

The `ask_enter_treasure(H)` interaction type holds for interactions `msg(H, bilbo, ask, enter_treasure)` where

H is the sender, who must be one among the hobbits, bilbo is the receiver, ask is the performative, enter_treasure is the content.

This request may be followed (sequence operator :) either by an interaction of type ok_enter(H) stating that H can enter the room, followed by a thanks by H, and then conclude (lambda), or (choice operator +) by Bilbo's refusal to let H in (no_enter(H)) followed by an expression of disappointment by H (grunt(H)), followed by either the hobbit branch protocol again, or lambda.

The BilboHobbits global type has as many different branches as the hobbits with whom Bilbo is expected to interact. The fork operator | is used to specify interleaving among interactions in these branches, which are also put in interleaving with (switch(bilbo,ef)^0:lambda) where msg(gandalf,R,switch,Pr) ∈ switch(R, Pr). This means that msg(gandalf,bilbo,switch,ef) is allowed to be received in any moment:

```
BilboHobbits =
(((Hobbit1Branch|Hobbit2Branch)|
   Hobbit3Branch)|...)|(switch(bilbo,ef)^0:lambda).
```

Once projected onto Bilbo, BilboHobbits returns BilboHobbits itself as Bilbo is involved in any interaction and none can be discarded. When projected onto hobbitj, BilboHobbits returns HobbitjBranch which drives the behavior of the j-th hobbit.

Exceptional Protocol. If an emergency takes place, Bilbo should ask Frodo to help him moving the treasure to a safer place. If Frodo can help Bilbo, he gives a positive answer, otherwise he asks Sam and Folco (no matter in which order). Only after these interactions take place (concatenation operator *), all the agents are ready to manage a new switch request from Gandalf, asking them to recover to their normal protocol. The ExceptionalFlow protocol, identified by ef, is specified by the following constrained global type:

```
ExceptionalFlow =
(ask_help(bilbo,frodo)^0:
   ((ok_help(frodo,bilbo)^0:lambda) +
    (cannot_help(frodo,bilbo)^0:
      (ask_help(frodo,sam)^0:lambda |
       ask_help(frodo,folco)^0:lambda))))*Recover,
Recover =
((switch(frodo,fr)^0:lambda) |
 (switch(bilbo,bh)^0:lambda) |
 (switch(sam,sa)^0:lambda) |
 (switch(folco,fo)^0:lambda))
```

Gandalf acts as the controller. The protocol that drives his behavior is

```
Gandalf =
((switch(frodo,ef)^0:lambda) |
 (switch(bilbo,ef)^0:lambda) |
 (switch(sam,ef)^0:lambda) |
 (switch(folco,ef)^0:lambda)) * Recover
```

where Recover is defined as in the ExceptionalFlow protocol. The decision of sending messages to Frodo, Bilbo, Sam and Folco is fired by the (paranormal) perception of a crowd of ogres coming near to the treasure room. After that (concatenation operator *) Gandalf sends a message to all the agents to switch back to their normal behavior. The constraint that when one protocol switch message is associated with the SwitchMsg variable, no other switch request can enter the message queue, is respected. In fact Frodo, Bilbo, Sam and Folco are able to manage switch requests at any time: as soon as they receive the first request by Gandalf they manage it and set SwitchMsg to null. When Gandalf sends the second request, they keep it in SwitchMsg until they complete the management of the exceptional situation and become ready to switch back to their normal life.

We experimented our framework on the above example testing different situations and observing the correct expected behavior in all of them. In particular, we made experiments where

– Bilbo lets the hobbits in;

– Bilbo does not allow the hobbits to enter the treasure room;

– Bilbo receives a protocol switch request from an agent which has no power to make it, and ignores it;

– Bilbo receives requests from the hobbits while he is executing the exceptional protocol, and ignores them;

– Frodo helps Bilbo;

– Frodo does not help Bilbo who involves Sam and Folco.

7. CONCLUSIONS AND FUTURE WORK

We have presented a framework for protocol-driven self-adaptive systems and its instantiation along the two following dimensions: formalism for representing protocols (constrained global types) and underlying agent's framework (Jason). The two main technical contributions of the paper are

1. the definition of protocol-driven agents and
2. the ability of changing behavior at runtime by switching from one protocol to another one; the protocol switch ability is a consequence of having protocols as first class entities.

The combination of features 1 and 2 with projection ensures that a protocol-driven agent Ag_1 whose behavior is driven by the protocol $\Pi(G, \{Ag_1\})$ obtained from projection by a protocol G, will be compliant by construction with all the agents driven by protocols $\Pi(G, \{Ag_2\})$, $\Pi(G, \{Ag_3\})$, ..., $\Pi(G, \{Ag_n\})$ that derive from G as well. Besides compliance to a global protocol, our approach supports the ability to adapt to new situations by protocol switching. These features seen together, and actually implemented on top of the Jason framework, contribute to the originality of our work.

In order to better assess the pros and cons of our proposal, we plan to select some standard scenario context, for example from e-commerce, and develop the same scenario using our proposed framework and using standard protocols with standard control actions (timeouts, exceptions, ...).

A positive aspect of our framework is that it should be possible to add it on top of all the agent environments and languages providing the basic building blocks for representing the agent knowledge, environment and message queue with a non negligible, but still limited, effort. We are working for verifying such a claim by instantiating our framework on JADE. A monitor implementing the *next* function on constrained global types has already been integrated in JADE [8]. We will start from that implementation to build the *generate* function and the protocol-driven interpreter around it.

Acknowledgments. This work has been partially supported by the MIUR PRIN Project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society, prot. 2010LHT4KM.

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, D. Briola, A. El Fallah Seghrouchni, V. Mascardi, and P. Taillibert. Efficient verification of MASs with projections. In F. Dalpiaz, J. Dix, and B. van Riemsdijk, editors, *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Revised Selected Papers*, volume 8758 of *Lecture Notes in Computer Science*. Springer, 2014.

[3] 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.

[4] M. Baldoni, C. Baroglio, and F. Capuzzimati. 2COMM: A commitment-based MAS architecture. In M. Cossentino, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Engineering Multi-Agent Systems*, volume 8245 of *Lecture Notes in Computer Science*, pages 38–57. Springer Berlin Heidelberg, 2013.

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

[6] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.

[7] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[8] D. Briola, V. Mascardi, and D. Ancona. Distributed runtime verification of JADE multiagent systems. In D. Camacho, L. Braubach, S. Venticinque, and C. Badica, editors, *Intelligent Distributed Computing VIII - Proceedings of the 8th International Symposium on Intelligent Distributed Computing, IDC 2014, Madrid, Spain, September 3-5, 2014*, volume 570 of *Studies in Computational Intelligence*, pages 81–91. Springer, 2014.

[9] N. Bulling, M. Dastani, and M. Knobbout. Monitoring norm violations in multi-agent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 491–498, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[10] 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.

[11] D. Capera, J. George, M.-P. Gleizes, and P. Glize. The AMAS theory for complex problem solving based on self-organizing cooperative agents. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 383–388, 2003.

[12] G. Casella and V. Mascardi. West2East: Exploiting WEb Service Technologies to Engineer Agent-Based SofTware. *Int. J. Agent-Oriented Softw. Eng.*, 1(3/4):396–434, 2007.

[13] G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multi-party session. *Logical Methods in Computer Science*, 8(1), 2012.

[14] L. Cernuzzi and F. Zambonelli. Dealing with adaptive multi-agent organizations in the Gaia methodology. In J. Müller and F. Zambonelli, editors, *Agent-Oriented Software Engineering VI*, volume 3950 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin Heidelberg, 2006.

[15] 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.

[16] 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.

[17] M. Cossentino. From requirements to code with the PASSI methodology. *Agent-oriented methodologies*, 3690:79–106, 2005.

[18] G. Cugola, C. Ghezzi, and L. Pinto. DSOL: a declarative approach to self-adaptive service orchestrations. *Computing*, 94(7):579–617, 2012.

[19] F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Adaptive socio-technical systems: a requirements-based approach. *Requirements Engineering*, 18(1):1–24, 2013.

[20] G. De Giacomo, Y. Lespérance, and C. Muise. On supervising agents in situation-determined ConGolog. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '12, pages 1031–1038, Richland, SC, 2012. International Foundation for Autonomous Agents and Multiagent Systems.

[21] G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-organization in multi-agent systems. *Knowl. Eng. Rev.*, 20(2):165–189, 2005.

[22] FIPA. FIPA ACL message structure specification. Approved for standard, Dec. 6^{th}, 2002, 2002.

[23] S. N. Gerard and M. P. Singh. Evolving protocols and agents in multiagent systems. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 997–1004, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems.

[24] 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.

[25] Z. Guessoum, M. Ziane, and N. Faci. Monitoring and organizational-level adaptation of multi-agent systems. In *Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on*, pages 514–521, 2004.

[26] C. Hahn, I. Zinnikus, S. Warwas, and K. Fischer. Automatic generation of executable behavior: A protocol-driven approach. In M.-P. Gleizes and J. Gomez-Sanz, editors, *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 110–124. Springer Berlin Heidelberg, 2011.

[27] IBM Corp. *An architectural blueprint for autonomic computing*. IBM Corp., USA, 2004.

[28] 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.

[29] 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.

[30] A. U. Mallya and M. P. Singh. Modeling exceptions via commitment protocols. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '05, pages 122–129, New York, NY, USA, 2005. ACM.

[31] V. Mascardi, D. Briola, and D. Ancona. On the expressiveness of attribute global types: The formalization of a real multiagent system protocol. In M. Baldoni, C. Baroglio, G. Boella, and R. Micalizio, editors, *AI*IA 2013: Advances in Artificial Intelligence - XIIIth International Conference of the Italian Association for Artificial Intelligence, Turin, Italy, December 4-6, 2013. Proceedings*, volume 8249 of *Lecture Notes in Computer Science*, pages 300–311. Springer, 2013.

[32] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In *ATAL*, pages 347–360. Springer Verlag, 1995.

[33] T. Miller and P. McBurney. Using constraints and process algebra for specification of first-class agent interaction protocols. In G. M. P. O'Hare, A. Ricci, M. J. O'Grady, and O. Dikenelli, editors, *Engineering Societies in the Agents World VII, 7th International Workshop, ESAW 2006, Dublin, Ireland, September 6-8, 2006 Revised Selected and Invited Papers*, volume 4457 of *Lecture Notes in Computer Science*, pages 245–264. Springer, 2006.

[34] T. Miller and P. McBurney. Annotation and matching of first-class agent interaction protocols. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, pages 805–812, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[35] T. Miller and P. McBurney. Propositional dynamic logic for reasoning about first-class agent interaction protocols. *Computational Intelligence*, 27(3):422–457, 2011.

[36] M. Morandini, F. Migeon, M.-P. Gleizes, C. Maurel, L. Penserini, and A. Perini. A goal-oriented approach for modelling self-organising MAS. In H. Aldewereld, V. Dignum, and G. Picard, editors, *Engineering Societies in the Agents World X*, volume 5881 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin Heidelberg, 2009.

[37] L. Padgham, J. Thangarajah, and M. Winikoff. AUML protocols and code generation in the Prometheus Design Tool. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS'07, pages 270:1–270:2, New York, NY, USA, 2007. ACM.

[38] 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. IEEE Computer Society, 2006.

[39] 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. IEEE Computer Society, 2007.

[40] R. de Lemos, H. Giese, H. A. Müller, et al. Software engineering for self-adaptive systems: A second research roadmap. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2013.

[41] 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.

[42] M. P. Singh. Interaction-oriented programming: Concepts, theories, and results on commitment protocols. In *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, AI'06, pages 5–6, Berlin, Heidelberg, 2006. Springer-Verlag.

[43] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White. A multi-agent systems approach to autonomic computing. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 464–471, Washington, DC, USA, 2004. IEEE Computer Society.

[44] S. Warwas and C. Hahn. The DSML4MAS development environment. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1379–1380, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[45] D. Weyns and M. Georgeff. Self-adaptation using multiagent systems. *Software, IEEE*, 27(1):86–91, 2010.

[46] Y. Brun, G. Di Marzo Serugendo, C. Gacek, et al. Engineering self-adaptive systems through feedback

loops. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.

[47] P. Yolum and M. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence*, 42(1-3):227–253, 2004.

[48] 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.

[49] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.