Typing Multi-Agent Systems via Commitments

Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati

Università degli Studi di Torino — Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)
{matteo.baldoni,cristina.baroglio,federico.capuzzimati}@unito.it

Abstract. This work presents an agent typing system, that differently than most of other proposals relies on notions that are typical of agent systems instead of relying on a functional approach. Specifically, we use commitments to define types. The proposed typing includes a notion of compatibility, based on subtyping, which allows for the safe substitution of agents to roles along an interaction that is ruled by a commitment-based protocol. Type checking can be done dynamically when an agent enacts a role. The proposal is implemented in the 2COMM framework and exploits Java annotations. 2COMM is based on the Agent & Artifact meta-model, exploit JADE and CArtAgO, by using CArtAgO artifacts in order to reify commitment protocols.

Keywords: Commitments, Static and dynamic type checking, Agents and Artifacts, JADE, Implementation

1 Introduction

Software infrastructures are quickly changing, becoming more and more global, pervasive and autonomic. Computing is becoming ubiquitous, with embedded and distributed devices interacting with each other. Multi-Agent Systems (MAS) have been recognized to be a promising paradigm for this kind of scenarios, however, as the complexity of programming these systems increases, the need for effective tools for reasoning on properties of programs becomes stronger and stronger. This is particularly true in the case of open systems, where heterogeneous and autonomously developed agents may need to interact. MAS usually rely on interaction protocols (or other kinds of "contract") to specify the interacting behavior that is expected of the agents. How can, then, an agent, a designer, the system verify that the agent has the the means for carrying on the encoded interaction? How to decide whether the agent is capable of behaving in a certain way or whether it shows specific skills/properties?

One way is to rely on some *typing of agents*, in a way that is similar to the typing of objects. Typing provides abstractions to perform sophisticated forms of program analysis and verifications: it helps performing compile-time/run-time error checking, modeling, documentation, verification of conformance and of compliance, reasoning about programs and components. It also allows a simple form of (a priori/runtime) verification. To the best of our knowledge, Zapf

and Geihs [34] were the first to propose the use of a type system for (mobile) agents, and they also introduced the idea of using sub-typing for the substitution of more specific subclasses in places where more general classes are expected, thus supporting safe extension and program re-use. More recent examples include [18,19,1,26]. In particular, [26] describes an agent-oriented programming language with a type checking that is inspired by mainstream object-oriented languages, and [1] uses global session types for realizing monitors of the interaction.

Differently than [18,19,26], we believe that, since types are abstraction tools for easily programming and modeling, for typing MAS it is necessary to rely on concepts that are typical abstractions of MAS, rather than relying on abstractions from other programming paradigms. Similarly to [1], our proposal is centered around *interaction*, which we believe to be one essential aspect of MAS. Differently than [1], we rely on commitments rather than on global session types. *Commitments* [13,28] are one of the fundamental abstractions for ruling agent interaction while preserving agent autonomy. For this reason, we discuss how commitments can be used for typing MAS and why it is interesting to rely on them. Specifically, we report the first steps towards a definition of a behavioral-based typing system for autonomous agents. The proposal is not bound to a specific agent programming language but, rather, it can be implemented in different frameworks. In the paper we describe an implementation in 2COMM [2]. The paper is organized as follows. Section 2 reports and comments the relevant literature motivating our proposal. Section 3 describes the 2COMM system that we used for the implemantation. Section 4 introduces the type system, while Section 5 describes its implementation. Conclusions end the paper.

2 Background and Motivation

The notion of "typing an agent" requires a precise, crisp definition. In programming languages, type systems are used to help designers and developers in avoiding code errors, bugs, that can entail unpredictable results. Type systems can be weak or strong, static or dynamic, but at the end they all share the same goal: support the development of error-free and human-readable code.

Most agent system implementations (JADE [9], Jack [20], A-Globe [29]) are based on programming languages like Java and do not supply agent type support but rather rely on the typing system of the language used for developing the system. Zapf and Geihs [34] underlined the importance of using a type system which allows dynamic type checking and proposed to base agent typing (1) on the externally visible actions of the agents, that they identify as being the messages agents accept and send, (2) on the meaning of the messages agents can exchange which includes, through the special symbol *self*, a characterization of the agent itself, (3) on the used communication protocol. They structure an agent type as a triple. The first component is the *syntactic type*, which is stateless and consists of the set of the input messages and of the set of output messages. The second is a *transition type*, i.e. a finite state automaton capturing a communication

protocol similarly to regular types [22]. The third and last component is the *semantic type*, an annotation aimed at checking behavior-compatibility, based on J. F. Sowa's conceptual graphs.

We agree on the importance of dynamic type checking for verifying that an agent fits the requirements for interacting in an open MAS in the moment the agent decides to enter the interaction, because it may have the required properties only when it enters the system; on the importance of relying only on externally visible actions, because the agents' internal states are not inspectable; on the importance of accounting for the interaction protocol, because it captures the rules of encounter of the agents, ruling their interaction. What we disagree with is the solution adopted by the authors of relying on finite state automata for describing the interaction as well as for describing the agents' behavior. This hinders the agent's autonomy in two ways. The first reason is that agents must supply a description of their behavior. Secondly, this description concerns how to do things, rather than what to do: it is prescriptive. An agent may have the possibility (and the capability) of doing something in different ways. We think that the typing system should be capable of featuring a more flexible representation of the behavior, with the possibility of leaving the choice of how to act up to the agent.

The main claim of [1] is the importance of using interaction protocols for representing the functioning of a system. To this aim, they use global session types as an abstraction tool, which allows automatically generating monitors that are aimed at verifying the correctness of on-going, multi-party interactions. In particular, the global session type is used to automatically generate a monitor agent, which intercepts all the exchanged messages and verifies whether the protocol is respected. This proposal is implemented in Jason [12]; a global session type is represented by a cyclic Prolog term, which is consumed as messages are sniffed. Along the line of the previous proposal, [1] focuses on externally visible actions (message exchanges) and on the use of interaction protocols. It differs from the previous one in that there is no actual type system, but rather global session types are used for specifying the interaction of a system from a global perspective. Since agents are not typed, when they enter a system, it is not possible to verify whether their behavior is compatible with the protocol nor it is possible to search for agents showing characteristics which allow them to successfully take part to the system. It is up to the monitor agent to check the exchanged messages. This is surely an important functionality but it is not type checking. In other words, the representation does not clearly express what an agent can do nor what is expected of an agent. Moreover, we disagree with the choice of realizing the monitor as an agent. In order for the system to be transparent, the monitor should be inspectable by the interacting agents, and the infrastructure should guarantee that the monitor is notified of all the exchanged messages. We believe that the environment should supply proper monitoring services, or an artifact, but not another autonomous agent.

Ricci and Santi [25,26] defined the SimpAL language, where types are seen as useful for realizing integrated development environments, and they imple-

mented an Eclipse plugin [27]. The approach to typing is a classic one, grounded on *interfaces*. This is the way in which most programming languages assure coherence, and prevent (statically) or detect (dynamically) logical errors. SimpAL extends the notion of interface to the agent abstraction level, introducing the notion of *role* as a collection of *tasks*, that an agent is capable to perform. A role will be implemented by an agent script, containing the behavioural logic of the agent. Specifically, a SimpAL *role* is an interface, while a role *task* is a method signature, which includes a list of formal parameters needed for its completion, that are expressed as pairs $\langle name : Type \rangle$. SimpAL provides environment typing and organizational typing too, used for programming coordination, resources and interactions between agents.

A typing of agents merely based on syntactic interfaces is criticized in [34], where the authors explain how conventional typing does not suffice the context of agent systems. The critic bases upon work by Nierstrasz [22] on active objects, that showed how the enumeration of the possible input and output messages is not sufficient to guarantee the interoperability. It is advisable to rely, instead, on some sort of behavioral type, including semantic information. Moreover, in SimpAL agent type checking is static. This is not a major concern in a homogeneous, single application environment. However, in an open MAS, where agents may be composed dynamically, static type checking is not enough; instead, it is necessary to rely on dynamic type checking and on monitoring. In this setting, agents themselves may verify their conformance to a role in order to decide whether to enter an interaction as well as to decide whether adopting new behaviors. As a consequence, the notion of type not only is a tool that supports the programmer's work but it becomes an programming element, that is used by agents in order to take decisions.

The proposal that we present in this paper concerns an agent typing system, which is characterized by (1) being based on typical agent society abstractions (social relationships), (2) being based on the agents' observable behavior, (3) dynamically checking if agents satisfy role requirements, (4) supplying a runtime monitoring environment. The implementation is provided in 2COMM, a middleware for developing open MAS whose interaction is commitment-based [2], which combines the well-known JADE [9] and CArtAgO [24] platforms. JADE agents interact based on commitment protocols. Each interaction protocol is realized as a CArtAgO artifact. Such an artifact provides social relationships as environmental resources. Dynamic checks are realized based on Java annotations.

3 Reference Framework

This proposal relies on the 2COMM middleware [2,3] for developing Multi-Agent Systems. In 2COMM, the MAS is specified as a set of *social relationships*, that govern the behavior of the agents taking part into the system. In a system made of autonomous and heterogeneous actors, social relationships cannot but concern the observable behavior [17]: for this reason, and in order to give them

that normative value which allows them to create social expectations, we realize social relationships by means of *commitments* [28].

On the other hand, we need social relationships to be *accepted* explicitly by the participants to the interaction, and possibly to be *inspected* by the agents, in order to decide whether conforming to them. To this aim, we need to explicitly model social relationships as resources, that are available to the interacting peers. Given that agents and social relationships are both first-class entities, that interact in a bi-directional manner, we adopt the Agents and Artifacts (A&A) meta-model [32,23], that extends the agent paradigm with another primitive abstraction, the artifact. A&A provides abstractions for environments and artifacts, that can be acted upon, observed, perceived, notified, and so on. When embodied inside artifacts, social relationships can be examined by the agents (to take decisions about their behavior), as advised in [14], used (which entails that agents accept the corresponding regulations), constructed, e.g., by negotiation, specialized, composed, and so forth.

2COMM[1] [2] provides a middleware for programming social relationships, by exploiting a declarative, interaction-centric approach. It is based on a combination of JADE [9] and CArtAgO [24]. JADE provides the agent platform, characterized by a FIPA compliant communication framework, and an agent-developing middleware. CArtAgO is a framework based on the A&A meta-model which extends the agent programming paradigm with the first-class entity of *artifact*: a resource that an agent can use. CArtAgO provides a way to define and organize *workspaces*, that are logical groups of artifacts, and that can be joined by agents at runtime. The environment is itself programmable and encapsulates services and functionalities. CArtAgO provides an API to program *artifacts* that agents can use, regardless of the agent programming language or the agent framework used. CArtAgO artifacts reify communication and interaction, represented in terms of commitment-based protocols. From an organizational perspective, a protocol is structured into a set of roles. A role represents a way of manipulating the social state and belongs to the artifact which reifies a protocol. Roles and agents are different entities, and we assume that roles cannot live autonomously: they exist in the system in view of the interaction, because agents, for interacting, use artifacts and execute actions on them [8]. Agents will use an interaction artifact to establish a channel of normed, mediated communication. The roles of such an artifact specify how agents can manipulate it: by enacting a role, an agent receives social powers by the artifact. Social powers have different and public social consequences, that we express in terms of commitments.

In 2COMM interaction is ruled by *commitment-based protocols*. A *commitment* $C(x, y, r, p)$ represents a directed obligation between a debtor x and a creditor y to bring about the consequent condition p when the antecedent condition r holds. A commitment may be manipulated by means of a set of primitives: delegate, assign, release [30]. They represents contractual relationships between agents, thus agents have the social expectation that an agent involved

[1] The source files of the system and examples are available at the URL `http://di.unito.it/2COMM`.

in a commitment as a debtor will realize the consequent condition; the debtor is responsible for the violation of a commitment. A *commitment protocol* defines a collection of actions, whose social effects are expressed in terms of commitment primitives, e.g., adding a new commitment, releasing another agent from some commitment, satisfying a commitment, see [33]. We assume that commitment conditions are yielded by the execution of artifact operations. For example, having a commitment C1 $= C(x, y, r, p \wedge q)$, a protocol artifact needs to supply at least an operation that makes r true, at least an operation that makes p true and at least an operation that makes q become true. The use of commitments gives a normative characterization to agent coordination [13,28]. When an agent uses a protocol artifact it accepts the regulations it contains and, in particular, that by executing certain actions it will be the debtor of some commitments. Public acceptance of the regulations is extremely important because it allows reasoning about the agents' behavior [15].

Figure 1 shows an excerpt of the 2COMM UML diagram. Overall the middleware is organized as follows: JADE supplies standard agent services (message passing, distributed containers, naming and yellow pages services, agent mobility); when needed, an agent can enact a protocol role, thus using a *communication artifact* – implemented by exploiting CArtAgO, which provides a set of operations by means of which agents participate in a mediated interaction session. Each communication artifact corresponds to a specific protocol enactment and maintains an own social state and an own communication state.

Class *CommunicationArtifact* (CA for short) provides the basic communication operations *in* and *out* for allowing mediated communication. by means of which agents respectively ask to play or to give up playing a role. CA extends an abstract version of the *TupleSpace* CArtAgO artifact: briefly, a blackboard that agents use as a tuple-based coordination means. In and out are, then, operations on the tuple space. CA also traces who is playing which role by using the property *enactedRoles*.

Class *Role* extends the CArtAgO class *Agent*, and contains the basic manipulation logic of CArtAgO artifacts. Thus, any specific role, extending this super-type, will be able to perform operations on artifacts, whenever its player will decide to do so. Role provides static methods for creating artifacts and for *enacting/deacting* roles. This is done by passing a reference to the JADE agent behavior that will actually play the role. The class *CARole* is an inner class of CA and extends the Role class. It provides the *send* and *receive* primitives, by which agents can exchange messages. Send and receive are implemented based on the *in* and *out* primitives provided by CA.

ProtocolArtifact (PA for short) extends CA and allows modeling the social layer with the help of commitments. It maintains the state of the on-going protocol interaction, via the property *socialState*, a store of social facts and commitments, that is managed only by its container artifact. This artifact implements the operations needed to manage commitments (create, discharge, cancel, release, assign, delegate). PA realizes the commitment life-cycle and for the assertion/retraction of facts. Operations on commitments are realized as *in-*

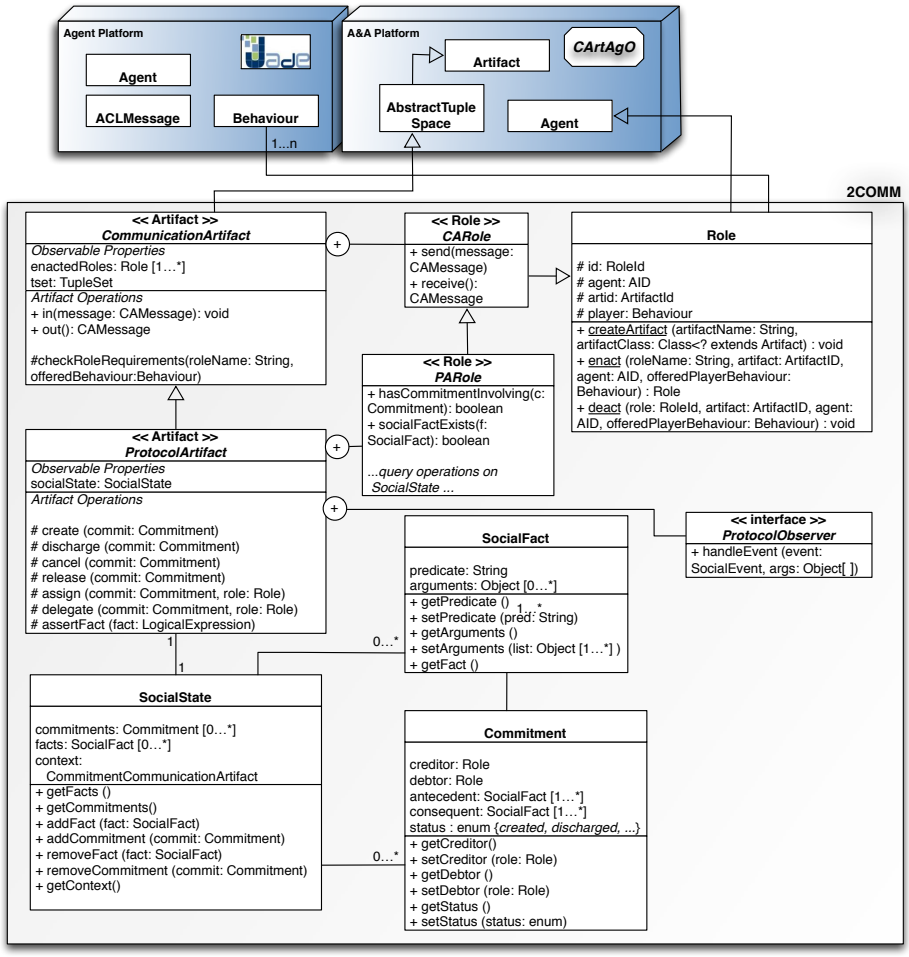

Fig. 1. UML Architecture of 2COMM.

ternal operations, that is, they are not invokable directly: the protocol social actions will use them as primitives to modify the social state. Being an extension of CA, PA maintains two levels of interaction: the social one (based on commitments), and the communication one (based on message exchange). The class *PARole* is an inner class of PA and extends the CARole class. It provides the primitives for *querying the social state*, e.g. for asking the commitments in which a certain agent is involved, and the primitives that allow an agent to become, through its role, an *observer of the events* occurring in the social state. For example, an agent can query the social state to verify if it contains a commitment with a specific condition as consequent, via the method

`existsCommitmentWithConsequent(InteractionStateElement el)`. Alternatively, an agent can be notified about the occurrence of a social event, provided that it implements the inner interface *ProtocolObserver*. Afterwards, it can start observing the social state. PARole also inherits the communication primitives defined in CARole.

In order to specify a *commitment-based interaction protocol*, it is necessary to extend PA by defining the proper social and communicative actions as operations on the artifact itself. Actions can have guards that correspond to *context preconditions*: each such condition specifies the context in which the respective action produces the described social effect. Since we want agents to act on artifacts only through their respective roles, when defining a protocol it is also necessary to create the roles. We do so by creating as many extensions of PARole as protocol roles. These extensions are realized as inner classes of the protocol: each such class will specify, as methods, the powers of a role. Powers allow agents who play roles to actually execute artifact operations. The reification of commitment protocols by way of artifacts has many advantages: by exploiting the distributed nature of artifacts it is possible to naturally rely on a modularization that helps the re-use of software, it is possible to implement run-time monitoring functionalities, and it is possible to provide a normative characterization of interaction thanks to commitments.

4 Typing MAS

To the aim of defining an agent typing system, we assume each agent a to be characterized by a set of behaviors $\{b_1, \ldots, b_m\}$, enabling a to perform various activities. Along the lines of [22], we view types as partial specifications of behavior, which support in using agents to play protocol roles safely. A type τ is a set of commitments $\{c_1, c_2, \ldots, c_n\}$, defined inside a collection of definitions of artifacts, that represents the environmental setting. The debtor, creditor, conditions of each commitment are defined as roles and actions inside some artifact, i.e. artifact definitions provide name spaces. Commitments, by having a normative value, can be seen as specifications of behavior because the debtor agents are expected to behave so as to satisfy them. A behavior b has type τ, denoted as $b : \tau$, if it is capable of satisfying the commitments in the type. This means that it allows to make the consequent conditions in the commitments become true.

Definition 1 (Type). *Given an agent a, with a set of behaviors $b_1 : \tau_1, \ldots, b_m : \tau_m$, we say that a has type $\tau = \bigcup_{i=1}^{m} \tau_i$, denoted as $a : \tau$.*

Let $P = r_1 \circ \ldots \circ r_n$ be an interaction protocol, where r_i are all the protocol roles. Let p be a protocol action, whose execution creates the commitments $c_1, \ldots, c_n$, (conditionally) binding the executor to achieve some conditions. This represents the fact that p *requires* the executor can satisfy (directly or indirectly – i.e. by way of other agents) $c_1, \ldots, c_n$. So, we say that p has type $\tau = \{c_1, \ldots, c_n\}$, denoted as $p : \tau$.

Definition 2 (Role and Protocol Types). *Let $p_1 : \tau_1, \ldots, p_m : \tau_m$ be the actions of P that the role r_j allows to execute together with their respective types. The type of role r_j is $\tau_j = \bigcup_{i=1}^{m} \tau_i$. Finally, the type of P is $\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$.*

We, now, introduce a notion of *subtype*, that is inspired to the width subtyping used for records. Given two types τ_1 and τ_2, we say that τ_1 is a *subtype* of τ_2, denoted by $\tau_1 \leq \tau_2$, when the set of commitments of τ_2 is included in the one of τ_1, i.e. $\tau_2 \subseteq \tau_1$. A subtype is a stronger specification which guarantees that the set of values satisfying it is a subset of the set of values of the supertype. What kinds of properties should types specify? According to the principle of substitutability [31] an instance of a subtype can always be used in any context in which an instance of the supertype is expected. A subtype at least guarantees the "promises" of the supertype, at least the same commitments, and possibly more, are satisfiable.

Since our subtyping relationship is defined based on subset inclusion, it is easy to see that subtyping is a *partial order*, and thus shows the properties of *reflexivity*, *antisimmetry*, and *transitivity*. More interestingly, the *subsumption* property also holds: consider an agent $a : \tau$ and suppose $\tau \leq \tau'$, then $a : \tau'$.

The rationale of the proposed subtyping relationship is that we mean to support the substitution of an actual agent and its behaviors to the specification of requirements that is given by a role: any behavior which is capable of achieving a superset of the required commitments will fit our case. Any operation feasible on the supertype will be supported by the subtype. This definition makes it possible to introduce a notion of *compatibility* of agents with roles.

Definition 3 (Compatibility). *An agent $a : \tau$ is* compatible with *a protocol role $r : \tau'$ if $\tau \leq \tau'$.*

In fact, since $a : \tau$ and $\tau \leq \tau'$, by subsumption $a : \tau'$. So, we are guaranteed that a can achieve the commitments it could get engaged into, when playing r, directly or by relying on other agents. Generally, a will have a more specialized behavior w.r.t. what the role demands.

We, now, show that subtyping guarantees substitutability: namely, that substituting a role by an agent that is compatible with it preserves the type of the protocol. Such a verification should be performed dynamically during the enactment of the protocol role.

Property 1 (Substitutability). Let $P = r_1 \circ \ldots \circ r_n$ be an interaction protocol of type τ. The system obtained by the enactment of the protocol, performed by the set of agents $a_1, \ldots, a_n$, each compatible with its respective P role, preserves the type τ.

The proof is trivially obtained by considering the above definitions.

Besides the behavioural-oriented notion of typing described above, we rely on Java to perform event (action) type checking. In fact, since they are implemented as artifact operations, when an agent uses an operation, through a role, the Java compiler checks the correctness of the parameters.

By adopting classical depth and width subtyping rules for records, i.e. $\{r_1 : \tau_1, \ldots, r_n : \tau_n\} \leq \{r_1 : \tau'_1, \ldots, r_m : \tau'_m\}$ if $m \leq n$ and $\tau_i \leq \tau'_i$, for all i from 1 to m, it is possible to introduce also a notion of protocol specialization.

Definition 4 (Specialization). *Let $P : \tau$ and $P' : \tau'$ be two interaction protocols with their respective types. We say that P' is a specialization of P if $\tau' \leq \tau$.*

5 Implementing the typing in 2COMM

Let us, now, introduce the way in which we implemented the proposed typing system in 2COMM. The implementation relies on *Java annotations*[2]. These are commonly used to provide meta-data about a program which can be used by the compiler, or be used at deploy time or, as in our case, at run-time.

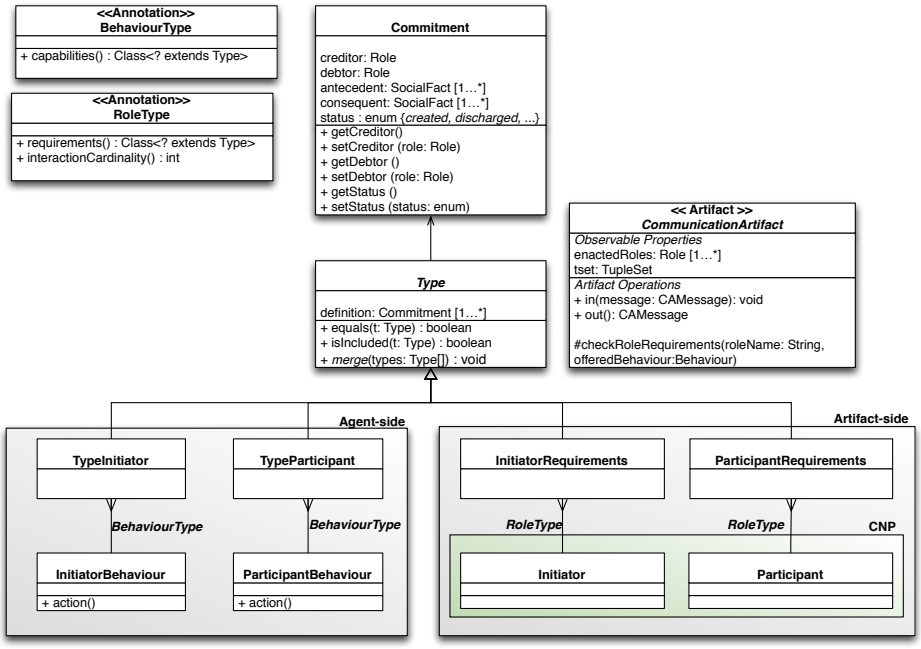

Fig. 2. UML Architecture of the typing system.

With reference to Figure 2, we introduced two annotations, one for interaction protocol roles, the other for agent behaviors. They are respectively *@RoleType*

[2] More information about Java annotations can be retrieved at http://docs.oracle.com/javase/tutorial/java/annotations/

and *@BehaviourType*. They both represent commitment sets. The former via the annotation property *requirements*, the latter via the annotation property *capabilities*. *@RoleType* also contains a property *interactionCardinality*, specifying whether a role can be concurrently played by many agents – as it is, for instance, the case of the Contract Net Protocol role Participant.

In our implementation, a type (Definition 1) is specified as an object of sort *Type*, which is an abstract class which contains the field *definition* (an array of commitments).

```
1  public abstract class Type {
2    final private ArrayList<Commitment> definition;
3    protected Type(Commitment[] commitsDefinition) {
4      definition = new ArrayList<Commitment>();
5      for (Commitment c : commitsDefinition) {
6        definition.add(c);
7      }
8    }
9    public boolean isIncluded(Type includerType)  {
10     boolean included = true;
11     for (Commitment c : this.definition) {
12       if (included) {
13         included = false;
14         for (Commitment d : includerType.definition) {
15           if (c.equals(d)) {
16             included = true;
17             break;
18           }
19         }
20       }
21       else break;
22     }
23     return included;
24   }
25   public boolean equals(Type t) {
26     return this.isIncluded(t) && t.isIncluded(this);
27   }
28   public static Type merge(ArrayList<Type> typesToMerge) {
29   ...
30   }
31   ...
32 }
```

Type must be subclassed by actual types, whose constructors will invoke the superconstructor and specify proper arrays of commitments. Moreover, *Type* specifies two methods, *equals* and *isIncluded* (that we report hereafter) which respectively verify if a type (set of commitments) is identical to another and if a type is subtype of another. A static, utility method *merge* is provided too, that creates a new Type object from the union of commitments of types passed as parameters.

The *equals* method considers two commitments equal if all their components are respectively equal.

```
1  public boolean equals(SocialStateElement el) {
2    if (el.getElType() != SocialStateElementType.COMMITMENT)
3      return false;
4    Commitment c = (Commitment)el;
5    return (this.getCreditor().equals(c.getCreditor()) &&
6      this.getDebtor().equals(c.getDebtor()) &&
7      this.getAntecedent().equals(c.getAntecedent()) &&
8      this.getConsequent().equals(c.getConsequent())
9    );
```

```
10 }
```

Antecedent and consequent formulas have to match exactly, while the identities of creditors and debtors are checked as follows:

```
1 public boolean equals(RoleId otherRoleId) {
2   if (this.type == otherRoleId.type && this.type == PARTICULAR_ROLE)
3     return this.id == otherRoleId.id;
4   else
5     return this.getRoleName().equals(otherRoleId.getRoleName());
6 }
```

The implementation can compare commitments that are instantiated and involve specific agents or that are "generic", in that they involve protocol roles. To separate the two cases, in the former the debtor and creditor of a commitment are associated to the case *PARTICULAR_ROLE* while in the latter they are associated to the case *GENERIC_ROLE*. This information is used by the method *equals*: A debtor/creditor identity is considered equal to that of another in two cases: (1) when the two refer to the very same enactment of a certain role (i.e. they refer to the same agent); (2) when one or both identities refer to a role type (e.g. the initiator) and the respective role names are equal.

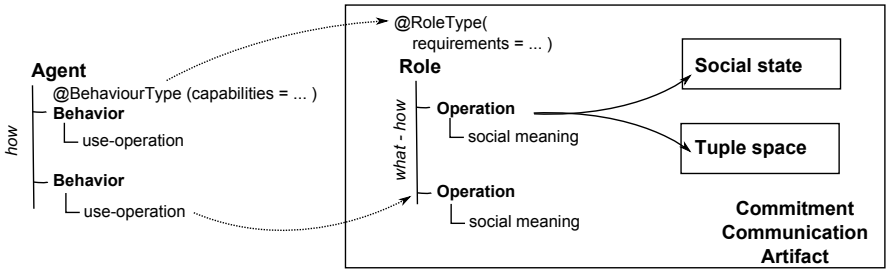

Fig. 3. Agent typing and roles definition.

With reference to Figure 3, type checking amounts to verifying if the commitments specified in the *capabilities* property of annotation @BehaviourType include the commitments specified in the *requirements* of the annotation @RoleType. The check is performed by the method *checkRoleRequirements* which is included in the class *CommunicationArtifact*. This method, which is executed in the context of *enactRole*, uses the set of behaviors of an agent and the role this means to play, and computes an answer by extracting at run-time the information contained in the involved annotations. An agent can successfully enact a role only if it is compatible with it (Definition 3), i.e. only if its type is a subtype of that of the role. For the property of substitutability, the enactment preserves the type of the protocol, thereby assuring safety.

```
1 public abstract class CommunicationArtifact extends AbstractTupleSpace {
2   ...
3   protected boolean checkRoleRequirements(String roleName,
```

```
4         Behaviour[] offeredPlayerBehaviours) {
5     // check the requested Role Name
6     if (!enabledRoles.containsKey(roleName)) {
7       logger.debug("Role "+roleName+" not found among enabled roles.");
8       return false;
9     }
10    // control is excluded for role "CA_Role"
11    if (roleName.equals(CA_ROLE))
12      return true;
13    Class<? extends Behaviour> behClass;
14    ArrayList<Annotation> behaviourTypeAnnotations
15              = new ArrayList<Annotation>();
16    Annotation behaviourSatisfyAnnotation;
17    for (Behaviour beh : offeredPlayerBehaviours) {
18      behClass = beh.getClass();
19      behaviourTypeAnnotation
20          = behClass.getAnnotation(BehaviourType.class);
21      if (behaviourTypeAnnotation == null)
22        // if null, correct annotation is missing
23        return false;
24    Class<?> roleClass;
25    try {
26      String roleClassName = (this.getClass().getName())
27          + "$" + roleName;
28      roleClass = Class.forName(roleClassName);
29    } catch (ClassNotFoundException e) {
30      return false;
31    }
32    Annotation roleAnnotation =
33        roleClass.getAnnotation(RoleType.class);
34    if (roleAnnotation == null) {
35      return false;
36    }
37    // Both annotations retrieved
38    // Getting instances for retrieved types
39    ArrayList<Type> typesToMerge = new ArrayList<Type>();
40    Type behaviourType;
41    Type roleType;
42    Type mergedType;
43    for (Annotation ann : behaviourTypeAnnotations) {
44      behaviourType = ((BehaviourType)ann).capabilities()
45                  .getDeclaredConstructor().newInstance();
46      typesToMerge.add(behaviourType);
47    }
48    roleType = ((RoleType)roleAnnotation).requirements()
49                .getDeclaredConstructor().newInstance();
50    mergedType = Type.merge(typesToMerge);
51
52    return roleType.isIncluded(mergedType);
53    }
54  }
```

When an agent tries to enact a role, the artifact, whose role is being enacted, is in charge for checking the compliance between the agent's behaviour and the role requirements. The method *checkRoleRequirements* of the class *CommitmentArtifact* performs these controls. This implementation realizes the principle of *compatibility*: an agent can enact a role provided it has a (set of) behaviour(s) that are compatible with the type of the role.

The *Type* abstract class, together with the *@RoleAnnotation* and *@BehaviourType* annotation classes, allows constructing types as Java structures, an approach similar to the one proposed in [34], where each agent carries an object representing its type.

Let us, now, show an example of annotation added on top of an implementation of the Contract Net Protocol presented in [3]. We will focus on the role Initiator and on an agent willing to play that role.

```
public class CNPArtifact extends ProtocolArtifact {
...
  @RoleType(requirements = InitiatorRequirements.class)
  public class Initiator extends PARole {
    public Initiator(Behaviour player, AID agent) {
      super(INITIATOR_ROLE, player, agent);
    }
    ...
  }
}
```

The role Initiator is tagged by the *@RoleType* annotation, whose value for the property *requirements* is set to *InitiatorRequirements.class*, a class that builds the set of commitments that defines the type of the role. *InitiatorRequirements* is specified in this way:

```
public class InitiatorRequirements extends Type {
  public InitiatorRequirements() throws MissingOperandException,
                                        WrongOperandsNumberException {
    super(new Commitment[]{
      new Commitment(CNPArtifact.INITIATOR_ROLE,
        CNPArtifact.PARTICIPANT_ROLE, "propose",
        new CompositeExpression(LogicalOperatorType.OR,
        new Fact("accept"), new Fact("reject")))
    });
  }
  ...
}
```

Specifically, this class contains the commitment $C(CNPArtifact.INITIATOR\_ROLE,\ CNPArtifact.PARTICIPANT\_ROLE,\ propose,\ accept \vee reject)$, where $CNPArtifact$ is the CommitmentArtifact which realizes the Contract Net Protocol.

On the agent's side, an agent willing to play the role *Initiator* must offer a set of behaviors that are typed accordingly. In our case, we suppose that the agent offers the following behavior:

```
@BehaviourType(capabilities = TypeInitiator.class)
public class InitiatorBehaviour extends OneShotBehaviour implements
        CNPInitiatorObserver {
    ....
}
```

where the class *TypeInitiator* specifies the *capabilities* shown by the agent through the behavior. Once again, this is a set of commitments the behavior can satisfy. *TypeInitiator* is a subclass the *Type*:

```
  public TypeInitiator() throws MissingOperandException,
        WrongOperandsNumberException {
    super(new Commitment[]{
      new Commitment(CNPArtifact.INITIATOR_ROLE,
          CNPArtifact.PARTICIPANT_ROLE, "propose",
          new CompositeExpression(LogicalOperatorType.OR,
          new Fact("accept"), new Fact("reject"))),
      new Commitment(TradeArtifact.BUYER_ROLE,
          TradeArtifact.SELLER_ROLE, "pay", "deliver"
          )
    });
```

```
12     }
13     ...
14 }
```

It is easy to see that the commitment perfectly matches the requirements, and so the enactment will succeed. Notice that the presented implementation is slightly different w.r.t. the definition of *compatibility* with a role (Definition 3): it uses a collection of behaviours instead of an agent because in JADE there is no reference to the agents that we could exploit. The result is a more restrictive test, which does not necessarily account for the whole agent but considers only the set of behaviors the agent displays.

6 Discussion and Future Work

This paper presented a typing system for MAS. The key characteristic of the proposal is that the typing system is defined based on notions that are typical of agents rather than on a functional approach. Specifically, it relies on the "social capabilities" of the agents. As such, the proposal represents a novelty w.r.t. previous work on agent typing, which applies the functional type theory [18,19,26]. The functional approach benefits of the results of a vast literature, but types should be aimed at providing abstraction/modeling features that help the programmer. Functional typing systems discard the typicalities of agents and, thus, in our view, they do not accomplish their aim.

Besides providing the basic notions of type, subtype, compatibility and substitutability, we implemented the proposal in the context of the 2COMM framework [2]. 2COMM allows programming social relationships by exploiting a declarative, interaction-centric approach, and was developed by relying on existing technologies as far as possible. In particular, the social relationships that arise along the interaction among agents are captured as social commitments – realized as first-class objects –, while interaction is mediated by protocol artifacts.

The choice of relying on commitments is motivated by the desire of typing agents and roles in a way that results minimally prescriptive, so to preserve the autonomy of the agents as far as possible. Indeed, we agree with [22,34] that the typing system should include a representation of the behavior but, differently than in those works – which deal with objects, we are also convinced of the need of a representation which does not hinder the agents' autonomy. For this reason, a prescriptive representation, based on finite state automata – as the one introduced in those works, would not be adequate. Commitments allow specifying the expected behavior of agents without imposing unnecessary restrictions. In case a more expressive language for specifying constraints is needed, it is possible to rely either on proposals like [21], where conditions inside commitments can express temporal regulations, or on proposals like 2CL [6], where commitment protocols are enriched with explicit temporal constraints on the evolution of the social state. This kind of extension is one of our next goals.

Clearly, a type system allows only a light check of the behavior of the involved agents, being more concerned with a safe usage rather than a full behavioral

compatibility. It does not imply that an agent which has the same type of another agent will display the same behavior. This does not exclude the possibility to integrate deeper checks, for instance based on model checking such as [10].

The described agent typing system will help realizing both static, compile-time coding support and dynamic, run-time type checking. Inspired by [27], the former can be realized by developing a plugin for an IDE that provides coding support, like smart code completion or type warning or error. The latter, instead, amounts to the development of tools for verifying, at run-time, the compliance between the agent's logics and the role requirements, signalling the occurrence of wrong enactments when needed. Altogether similar tools based on the substitutability property, which guarantees the safe replacement of agents to roles, when they have the same type or the agent has a subtype of the role. In the current proposal such a verification is performed as a syntactic inclusion of commitment sets. This is limitating because it does not consider logical expressions inside commitment antecedent and consequent conditions. To solve the problem we mean to study the applicability of complex typing systems, relying on union and intersection types [16].

Type checking as a light verification adopts notions, e.g. substitutability, that are used also for facing the issues of interoperability and conformance discussed in [7,5]. The conformance verification aims at guaranteeing that when an agent plays a role, or substitutes another agent in an on-going interaction, the interoperability of the system is preserved – in the present paper, when an agent plays a role, the protocol type is preserved. In [7,5], protocols are represented by way of a sort of finite state automata. Thus, the approach suffers from the drawbacks due to a prescriptive description, that, as we explained (Section 2, see the comments to the approach in [1]), does not suit well the autonomy of the agents. Another direction of research that we mean to pursue is to explore how commitment-based types can be adapted to solve the issue of conformance in MAS.

Finally, in [4], we presented an extension of JaCaMo [11] that, analogously to 2COMM, allows reasoning about social relationships in Jason agents. We aim to introduce the use of the proposed typing system also in that setting. This would allow an even deeper comparison to SimpAL, which is built on top of the same platform.

Acknowledgements

We thank the anonymous reviewers for their helpful comments, which gave us important suggestions for future developments.

References

1. Davide Ancona, Sophia Drossopoulou, and Viviana Mascardi. Automatic generation of self-monitoring mass from multiparty global session types in jason. In Matteo Baldoni, Louise A. Dennis, Viviana Mascardi, and Wamberto Vasconcelos,

editors, *DALT*, volume 7784 of *Lecture Notes in Computer Science*, pages 76–95. Springer, 2012.

2. M. Baldoni, C. Baroglio, and F. Capuzzimati. 2COMM: a commitment-based MAS architecture. In M. Cossentino, A. El Fallah Seghrouchni, and M. Winikoff, editors, *Post-Proc. of the 1st International Workshop on Engineering Multi-Agent Systems, EMAS 2013, Revised Selected and Invited Papers*, number 8245 in LNAI, pages 38–57. Springer, 2013.
3. Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. A Commitment-based Infrastructure for Programming Socio-Technical Systems. *ACM Transactions on Internet Technology, Special Issue on Foundations of Social Computing*, 2014. To appear.
4. Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. Reasoning about Social Relationships with Jason. In Amit Chopra and Harko Verhagen, editors, *Proc. of the 1st International Workshop on Multiagent Foundations of Social Computing, SC-AAMAS 2014, held in conjuction with AAMAS 2014*, Paris, France, May 2014.
5. Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. Choice, Interoperability, and Conformance in Interaction Protocols and Service Choreographies. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009*, pages 843–850. IFAAMAS, 2009.
6. Matteo Baldoni, Cristina Baroglio, Elisa Marengo, and Viviana Patti. Constitutive and Regulative Specifications of Commitment Protocols: a Decoupled Approach. *ACM Transactions on Intelligent Systems and Technology, Special Issue on Agent Communication*, 4(2):22:1–22:25, March 2013.
7. Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. A priori conformance verification for guaranteeing interoperability in open environments. In A. Dan and W. Lamersdorf, editors, *Proc. of the 4th International Conference on Service Oriented Computing, ICSOC 2006*, volume 4294 of *LNCS*, pages 339–351, Chicago, USA, December 2006. Springer.
8. Matteo Baldoni, Guido Boella, and Leon van der Torre. Interaction between Objects in `powerjava`. *Journal of Object Technology, Special Issue OOPS Track at SAC 2006*, 6(2), 2007.
9. F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - A Java Agent Development Framework. In R. H. Bordini, M. Dastani, J. JDix, and A. El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer, 2005.
10. J. Bentahar, J.-J. Ch. Meyer, and W. Wan. Model Checking Communicative Agent-based Systems. *Knowledge-Based Systems*, 22(3):142–159, 2009.
11. Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ric ci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747 – 761, 2013.
12. RafaelH. Bordini and JomiF. Hübner. Bdi agent programming in agentspeak using jason. In Francesca Toni and Paolo Torroni, editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg, 2006.
13. C. Castelfranchi. Principles of Individual Social Action. In G. Holmstrom-Hintikka and R. Tuomela, editors, *Contemporary action theory: Social action*, volume 2, pages 163–192, Dordrecht, 1997. Kluwer.

14. Amit K. Chopra and Munindar P. Singh. Elements of a business-level architecture for multiagent systems. In Lars Braubach, Jean-Pierre Briot, and John Thangarajah, editors, *PROMAS*, volume 5919 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2009.
15. Rosaria Conte, Cristiano Castelfranchi, and Frank Dignum. Autonomous norm acceptance. In Jörg P. Müller, Munindar P. Singh, and Anand S. Rao, editors, *ATAL*, volume 1555 of *Lecture Notes in Computer Science*, pages 99–112. Springer, 1998.
16. Mario Coppo, Mariangiola Dezani-Ciancaglini, Ines Margaria, and Maddalena Zacchi. Toward isomorphism of intersection and union types. In Stéphane Graham-Lengrand and Luca Paolini, editors, *ITRS*, volume 121 of *EPTCS*, pages 58–80, 2013.
17. Mehdi Dastani, Davide Grossi, John-Jules Ch. Meyer, and Nick A. M. Tinnemeier. Normative Multi-agent Programs and Their Logics. In John-Jules Ch. Meyer and Jan Broersen, editors, *KRAMAS*, volume 5605 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2008.
18. Claudia Grigore and Rem Collier. Supporting agent systems in the programming language. In Jomi Fred Hübner, Jean-Marc Petit, and Einoshin Suzuki, editors, *Web Intelligence/IAT Workshops*, pages 9–12. IEEE Computer Society, 2011.
19. Claudia Grigore and Rem W. Collier. Af-raf: an agent-oriented programming language with algebraic data types. In Cristina Videira Lopes, editor, *SPLASH Workshops*, pages 195–200. ACM, 2011.
20. Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Intelligent agents - summary of an agent infrastructure. In *Proc. of the 5th International Conference on Autonomous Agents*, 2001.
21. Elisa Marengo, Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Viviana Patti, and Munindar P. Singh. Commitments with Regulations: Reasoning about Safety and Control in REGULA. In K. Tumer, P. Yolum, L. Sonenberg, and P. Stone, editors, *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2011*, volume 2, pages 467–474, Taipei, Taiwan, May 2011. IFAAMAS.
22. Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*, chapter 6, pages 99–121. 1995. Prentice Hall.
23. Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.
24. Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.
25. Alessandro Ricci and Andrea Santi. From actors to agent-oriented programming abstractions in simpal. In Gary T. Leavens, editor, *SPLASH*, pages 73–74. ACM, 2012.
26. Alessandro Ricci and Andrea Santi. Typing Multi-agent Programs in simpAL. In Mehdi Dastani, Jomi Fred Hübner, and Brian Logan, editors, *ProMAS*, volume 7837 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2012.
27. Andrea Santi and Alessandro Ricci. An eclipse-based ide for agent-oriented programming in simpal. In *Proc. of The Seventh Workshop of the Italian Eclipse Community*, 2012.
28. Munindar P. Singh. An ontology for commitments in multiagent systems. *Artif. Intell. Law*, 7(1):97–113, 1999.

29. David Šišlák, Martin Rehák, Michal Pěchouček, Milan Rollo, and Dušan Pavlíček. A-globe: Agent development platform with inaccessibility and mobility support. In *Software Agent-Based Applications, Platforms and Development Kits*, pages 21–46. Birkhäuser Basel, 2005.
30. Pankaj R. Telang and Munindar P. Singh. Specifying and Verifying Cross-Organizational Business Models: An Agent-Oriented Approach. *IEEE Transactions on Services Computing*, pages 1–14, 2011.
31. Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *Proceedings ECOOP '88*, number 322 in Lecture Notes in Computer Science, pages 55–77. Springer-Verlag, 1988.
32. Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.
33. Pınar Yolum and Munindar P. Singh. Commitment Machines. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001*, volume 2333 of *LNCS*, pages 235–247. Springer, 2002.
34. Michael Zapf and Kurt Geihs. What type is it? a type system for mobile agents. In *15th European Meeting on Cybernetics and Systems Research (EMCSR)*, 2000.