Type checking for protocol role enactments via commitments

Noname manuscript No.
(will be inserted by the editor)

Type Checking for Protocol Role Enactments via Commitments

Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio

the date of receipt and acceptance should be inserted later

Abstract This work presents a commitment-based agent typing system. Type checking is done dynamically when an agent enacts a commitment-based protocol role: verification checks if the agent meets the requirements displayed by the role it means to enact. An example implementation in the 2COMM4JADE framework is provided. 2COMM4JADE is based on the Agent & Artifact meta-model and exploits JADE and CArtAgO, by using CArtAgO artifacts in order to reify commitment protocols.

Keywords Agent Typing, Social Relationships, Static and dynamic type checking, Commitments, Commitment-based Interaction Protocols.

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 because a MAS may be composed of heterogeneous and autonomously developed agents, which need to operate in a same environment, share resources, contribute to the achievement of goals.

In this context, *interaction*, which is recognized as one of the main dimensions that make a MAS [35], becomes an essential aspect. MAS generally 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 designer verify that an agent has the means for carrying out the encoded interaction? How to decide whether the agent is capable of behaving in a certain way, or whether it shows specific skills/properties?

A practical way, yet grounding on solid methodological bases, 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 verification: it helps performing compile-time/run-time error checking, modeling, documentation, verification of confor-

Università degli Studi di Torino, Dipartimento di Informatica, c.so Svizzera 185, I-10149 Torino (Italy).
E-mail: firstname.lastname@unito.it

mance and of compliance, reasoning about programs and components. It allows light forms of (a priori/runtime) component verification. The research question is then which characteristics should be considered in the conceptualization and realization of a typing system for interacting agents.

First of all, since the typing we are interested in is about agents, it is fundamental that such a typing be *homogeneous with the agent paradigm*. It should therefore rely on notions that are the basic building blocks of the agent paradigm, such as those of behaviors and goals. This would make the typing system more natural to a programmer who would not need to mix agent concepts with concepts from, e.g. a functional perspective that bases on totally different grounds.

Then, in order to accommodate the agents' autonomy, the most peculiar feature of the agent abstraction, the typing system should be as least prescriptive as possible. One way of meeting this requirement is that the typing system be *declarative*. A declarative approach, in fact, should only state *what* a successful interaction should look like, without imposing *how* the interaction should be carried out. A typing system which is declarative in nature results to be more flexible from the agents' point of view, and hence supports their autonomy.

A further essential characteristic of agents is that they may acquire or lose capabilities along time. Consequently, a typing system for interacting agents should allow *dynamic type checking*, which should be triggered just before an agent tries to join an interaction. Second, as discussed in [71], a static type checking would be in general unpractical because it should verify whether all the possible sequences of messages an agent can send and receive are appropriate for joining an interaction.

Finally, since agents can enact roles in a protocol in a distributed way, also the type checking, occurring right at enactment time, must be *distributed*, or local, and rely on information possessed by the agent.

The agent typing presented in this work possesses the characteristics described above, and to the best of our knowledge, no agent typing system having all these characteristics has yet been proposed in literature. It basically concerns agent interaction, and has the ultimate purpose of allowing the verification that agents satisfy the requirements for the protocol roles they mean to enact. The check is performed dynamically at *role-enactment time* to account for the fact that agents capabilities may change along time. The important thing is, in fact, that the needed capabilities are available at the right moment, i.e. when the agent starts playing a role. Building upon a wide literature that considers relationships as one of the basic building blocks of the human way of interpreting reality[1], our typing system is conceptually centered around those *social relationships*, which agents may create along their interaction when playing *protocol roles*, because such relationships create *expectations* on the agents behavior.

Contribution and Organization. The major contribution of this paper is an agent-based, dynamic, and declarative type checking system for agent interactions modeled via commitment-based protocols. To this end, the paper provides in Section 2 a synthetic description of background notions regarding commitments, commitment-based protocols, and protocol roles and role enactment. Social commitments [24,59] are, indeed, one of the fundamental abstractions for ruling agent interaction, while preserving agent autonomy. We discuss (Section 3) how commitments can be used for typing MAS and why it is interesting to rely

[1] The other one being object, a perspective that is widely reflected in computer science proposals – just think to the entity-relationship model.

on them. In particular, to meet the declarative requirement, commitment conditions are expressed in terms of precedence logic, a temporal logic introduced in [61] and in [62, Chapter 14] for Web service composition, that is sufficiently powerful to model dependencies among agents' behaviors (e.g., choice, parallelism, and precedence), hiding execution details such as metric temporal constraints (e.g., deadlines as proposed in [43], which can however be accounted for, as explained in Section 2). In the same section we show how the typing system built upon precedence logic enjoys an important progression property: when an agent joins an interaction, it is guaranteed that it possesses all the required behaviors to conduct its part of the interaction as far as one of the interaction acceptance states. We therefore give a formal characterization of a class of commitment-based protocols that supports the proposed typing (socially-progressive protocols), and formally define and characterize the notions of debtor and creditor compliance at the basis of the progression property.

Notably, the proposal is not bound to a specific agent programming language; on the contrary, it can be implemented in different frameworks. A further contribution (Section 4) is an exemplification of the approach in a real programming framework. We explain how the typing system and the checking performed at enactment-time were implemented in 2COMM4JADE [6], providing and discussing examples.

This paper significantly improves and extends the proposal in [7], where temporal, precedence logic was not considered. Moreover, the paper represents an important step forward compared to previous approach discussed in literature. Section 6 provides a thorough account of the current solutions to the problem of type checking agent-based programs, and how this proposal overcomes their shortcomings.

2 Background

The typing system we present is specifically conceived for multi-agent systems where agents need to interact to bring about their own goals. It is therefore quite natural to build the typing system on top of the same abstractions that are at the basis of agent society models. In our proposal, in particular, the most relevant notions are those of social relationship and role. Both will be considered as first-class entities. The conceptual framework of the proposal builds upon [6], where social relationships are explained to be a key component in the representation of socio-technical systems. The key aspects of that proposal are:

- Interaction protocols specify "what" should occur rather than "how" making it occur (minimal critical specification principle [25]);
- Agents share a (notional) social state of their interaction, containing an explicit representation of the social relationships that tie one agent to another;
- Social relationships:
  - have a normative force, they are accepted explicitly by the participants to the interaction;
  - agents can inspect them to decide whether conforming to them;
  - are resources, that are made available to the interacting agents and can be manipulated by them;
  - concern the observable behavior;
  - are modeled as *social commitments*.

We now provide a short introduction about commitments and commitment-based protocols.

2.1 Social Commitments

We represent social relationships among agents in terms of *social commitments* [59]. A social commitment models the directed relation between two agents: a *debtor* and a *creditor*. A commitment $C(a_1, a_2, p, q)$ captures that agent a_1 (debtor) commits to agent a_2 (creditor) to bring about the consequent condition q when the antecedent condition p holds. Along the line of [46, 12], commitments are defined over temporal expressions in *precedence logic* [61], [62, Chapter 14]. This logic is an event-based linear temporal logic thought to specify constraints on events of different services to be composed. The interpretation of such a logic deals with occurrences of events along runs (i.e., sequence of instanced events). Under this respect, event occurrences are assumed as nonrepeating and persistent: once an event has occurred, it has occurred forever. However, the specification of temporal expressions is done on "event literals", namely symbols that constitute the universe of discourse, and that are instanced by event occurrences along runs. This precedence logic has three primary operators: '$\vee$' (choice), '$\wedge$' (concurrence), and '$\cdot$' (before). The *before* operator enables one to express conditions such as $pay \cdot deliver$: both *pay* and $deliver$ must occur and in the specified order, but the two events do not need to occur immediately after one another. The language also includes event complementation, which can be thought of as a form of negation for events. Let e be an event. Then $\overline{e}$, the complement of e, is also an event. Initially, neither e nor $\overline{e}$ hold. On any run, either e or $\overline{e}$ may occur, not both. Intuitively, complementary events allow one to specify situations in which an expected event e does not occur, either because of the occurrence of an opposite event, or because of the expiration of a time deadline. For instance, the occurrence of event *confirm* in a protocol could be considered as the confirmation of a previous proposal that an agent communicates to another agent. The complementary event $\overline{confirm}$ could be the effect of an explicit cancellation by the agent, or of a timeout (e.g., if the agent does not confirm within a specific time interval, a cancellation is assumed). Complementation is extended to expressions in general. Given temporal expressions p and q, (i) $\overline{p \wedge q} = \overline{p} \vee \overline{q}$. (ii) $\overline{p \vee q} = \overline{p} \wedge \overline{q}$. (iii) $\overline{p \cdot q} = \overline{p} \vee \overline{q} \vee (q \cdot p)$. For simplicity, the events can be thought of as propositional but can be generated schematically as in *pay*\$1, *pay*\$2.

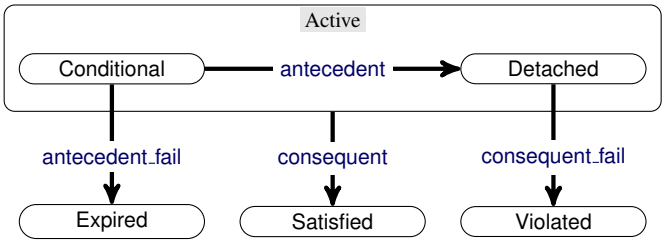

Fig. 1: Commitment life cycle.

Commitment evolution follows the life cycle formalized in [67], which is reported in Figure 1. A commitment is *Violated* either when its antecedent is true but its consequent will forever be false, or when it is canceled when Detached. It is *Satisfied*, when the engagement is accomplished (notice that a commitment does not need to be detached before being satisfied). It is *Expired*, when it is no longer in effect and therefore the debtor would not fail to comply even if does not accomplish the consequent. Active has two substates: *Conditional* as long as the antecedent does not occur, and *Detached* when the antecedent has occurred.

Commitments have a *normative value* because the debtor of a Detached commitment is expected to bring about, sooner or later, the consequent condition of that commitment otherwise it will be liable for a violation. Thus, commitments provide social expectations on the agents' behavior. A commitment is *autonomously taken* by a debtor towards a creditor on its own initiative and it is manipulated through the standard operations *create*, *detach*, *expire*, *discharge*, *violate*, *assign*, *delegate*, *release*, *cancel*. Create generates a new commitment, detach amounts to the satisfaction of the antecedent, expire to the falsification of the antecedent, discharge to the dischargement of the commitment, violate to the falsification of the consequent of a detached commitment, assign shifts a commitment to another creditor, delegate shifts a commitment to another debtor, release is used by a creditor to removes a commitment towards itself, cancel by a debtor to remove a commitment of its own. We do not tackle assign and delegate in this work.

Example 1 The commitment $C(merchant, client, pay, pack \cdot deliver)$ captures that an agent, playing the role *merchant*, committed to another agent, playing the role *client*, to deliver some item at a specified address if the client agent paid. The occurrence of *pay* detaches the commitment, according to the commitments life cycle in Figure 1. Afterwards, the *merchant* will have an obligation to first pack the item and, then, deliver it. When $pack \cdot deliver$ occurs, the commitment is satisfied. ■

Notice that in the understanding of commitments we refer to [59], *merchant* can pack and deliver an item even before payment occurs (e.g. when they trust the *client* or for some reason decide to give the item for free). The commitment would be satisfied also in this case. Moreover, the *merchant* may pack the item, do something else (such as show some other item to the client or answer to questions), and only after perform the delivery. It is also not required that *merchant* herself delivers the item: the delivery could, for instance, be performed by some shipper hired by *merchant*. What is important here is that *merchant* is liable in case the delivery does not occur when the payment was made as well as when the item is delivered without packing it.

Commitments arise, exist, are satisfied or otherwise manipulated all within a given *social context*. Not only they rely on the social structure of the groups within which they exist, but also help create that structure [59, 12]. Such a context can be identified as the group of agents (organization or institution) within which the agents interact.

2.2 Commitment-Based Protocols

Commitment-based protocols, as introduced in [60, 70, 69], capture the intrinsic meaning of actions in terms of how they create and manipulate commitments. They were introduced with the aim of overcoming the limits of approaches that specified protocols as sequences of allowed interactions; namely, the difficulty in accommodating the key aspects of autonomy, heterogeneity, opportunities, and exceptions, which are crucial to realize the open, dynamic nature of interactions. Commitment-based protocols assume that a (notional) *social state* is available and inspectable by all the involved agents. The social state traces which commitments currently exist, together with their states (according to the lifecycle and the interaction occurred so far), and with other facts that are relevant to the interaction.

As in [30], we see commitments (and also the social state) as "institutional objects", i.e. abstract objects that are created and manipulated according to a set of conventional rules, on which there is collective agreement by a community of agents. However, relying on works like [21, 65, 40, 47], that explain how social relationships (hence, commitments) can

be fully leveraged only when the notion of *role* becomes a first-class conceptual entity as well, we refine this vision by adopting the ontological definition of role that was proposed by Boella and van der Torre [21][2]. This is based on the organization metaphor: organizations, more in general institutions, can be manipulated only by following the rules that they themselves defined, and that are offered only to those agents which play a role within the institution. The authors point out that roles, by belonging to social institutions such as organizations, are actually social roles, and isolate three main properties relating roles, their players, and institutions: (1) a role must always be associated with the institution it belongs to and with its player; (2) a role is meaningful only within the institution it belongs to, and hence the definition of a role is only given within an institution; (3) the actions defined for a role in an institution have access to the state of the institution and of other roles; these actions are, therefore, the *powers* a role player is endowed with. Thus, in our understanding a commitment-based interaction protocol amounts to a *set of role powers*, that are specified in terms of how they manipulate commitments. Each commitment protocol role provides to its role players the powers of affecting the social state of interaction by executing the actions associated to that role.

More formally, let $\mathscr{B}$ be a nonempty set of events. Let $\mathscr{E}$ be the set of event temporal expressions generated from $\mathscr{B}$. Let $\mathscr{C}$ be a set of possible commitments of the form $C(x,y,p,q)$, where x and y are roles, while p and q are temporal expressions in $\mathscr{E}$, Let $\mathscr{S}$ be the set of possible operations on commitments.

Definition 1 A *protocol* is a tuple $\langle A, \rho, power, I\rangle$, where:

- A is the set of protocol actions. Each $a \in A$ is a pair $\langle e, E\rangle$ where $e \in \mathscr{B}$ is the action name, including its parameter list when needed, E is a set of possible operations on commitments in $\mathscr{S}$, that captures the effect of the action execution over the social state;
- ρ is a set of role names, identifying the interacting parties;
- $power : \rho \to 2^A$ is a mapping that associates each role with the set of actions empowered to that role. For each role name $x \in \rho$, $power(x)$ denotes the subset of protocol actions in A that any player of x can perform in P. So, $power(x)$ is the set of powers that are endowed to any agent enacting x;[3]
- I is a set of commitments that are active at the beginning of the interaction, and that are thus contained in the social state.

For practical reasons, we denote by $\mathscr{C}_P$ the subset of $\mathscr{C}$ of the commitments that can possibly be generated along the executions of protocol P, and by $\mathscr{B}_P$ the set of events that can possibly be generated along the execution of P.

Example 2 The Contract Net Protocol (CNP) involves two roles: *initiator* and *participant*. An agent playing the *initiator* role calls for proposals from any other agent playing the role of *participant*. A *participant* makes a proposal if interested. Proposals are either accepted or rejected by the *initiator*. Table 1 reports the powers of the *initiator* and *participant* roles, and their social effects, i.e., how these actions affect the social relationships between the involved agents. *Cfp* has, as social meaning, the creation of a commitment $C(i,p,propose,accept \vee \overline{accept})$, by which the *initiator* commits to answer to a participant's proposal either with acceptance or non-acceptance. On the other hand, *propose* has, as social meaning, the creation of a commitment $C(p,i,accept,done \vee \overline{done})$, by which the

[2] See [15–17] for a declination of the proposal in an Object-Oriented context.

[3] This means that actions have a social effect only if they are executed by an agent who is playing a role the action is associated with. It does not mean that an agent is enforced to execute only the actions associated with its role.

participant commits to carry out the assigned task to completion (*done*) or to failure ($\overline{done}$). Notice that the execution of *accept* has the twofold consequence to satisfy the commitment created by *cfp* and to detach the one created by *propose*, while $\overline{\overline{accept}}$ satisfies the former commitment and causes the expiration of the latter. ■

Table 1: The CNP protocol definition.

ROLES	POWERS	SOCIAL EFFECTS
INITIATOR:	*cfp*	create(*C(i, p, propose, accept* $\vee \overline{accept}$*)*)
	accept	–
	$\overline{accept}$	–
PARTICIPANT	*propose*	create(*C(p, i, accept, done* $\vee \overline{done}$ *)*)
	$\overline{propose}$	–
	done	–
	$\overline{done}$	–

Example 3 An alternative representation of CNP, reported in Table 2, is to rely on an explicit use of the *release* commitment operation. In particular, *reject* has as social meaning the release of the *participant*'s commitment, that was created when the latter executed *propose*. It also satisfies of the commitment created on the execution of *cfp*. *Accept*, *done*, and *failure*, instead, impact on the progression of commitment states along the lifecycle. For instance, *accept* causes the satisfaction of the commitment created by *cfp*. ■

Table 2: The CNP protocol definition.

ROLES	POWERS	SOCIAL EFFECTS
INITIATOR	*cfp*	create(*C(i, p, propose, accept* $\vee$ *reject)*)
	reject	release(*C(p, i, accept, done* $\vee$ *failure)*)
	accept	–
PARTICIPANT	*propose*	create(*C(p, i, accept, done* $\vee$ *failure)*)
	refuse	release(*C(i, p, propose, accept* $\vee$ *reject)*)
	done	–
	failure	–

2.3 Commitment Semantics

Concerning the semantics of commitments with temporal expressions, we rely on the proposal introduced in [46], later extended to include dialectical commitments [12]. Briefly, the semantics is given in terms of a model $M = \langle \mathbb{E}, \mathbb{T}, \mathbb{C}, \mathbb{D}, \mathbb{X}, \mathbb{V} \rangle$, where $\mathbb{E}$ is a denumerable set of possible events, while $\mathbb{T}$ is the set of possible event runs. Intuitively, $\mathbb{C}$ determines which commitments are active (i.e. conditional or detached) from a debtor to a creditor at an index in a run. $\mathbb{C}$ does so by assigning a set of runs to a set of runs, at each index on each run, for each debtor-creditor (ordered) pair of agents. $\mathbb{D}, \mathbb{X}, \mathbb{V}$ are respectively the standards for discharged, expired, and violated practical commitments.

We assume semantic postulates M1–M7 from [46]. For example, the semantic postulate M6 accommodates active commitments:

$$\tau \models_i C^p(x,y,p,q) \textit{ if and only if } [\![q]\!] \in \mathbb{C}_{x,y}(\tau, i, [\![p]\!])$$

Here, we denote by $[\![q]\!]$ the *intension* of q, that is, the set of runs where q is true on index 0: $[\![q]\!] = \{\tau \mid \tau \models_0 q\}$; $\tau_{[i,j]}$, instead, refers to the projection of τ from index i to j, both inclusive. Intuitively, a commitment is active at index i of run τ iff the intension of its consequent condition q belongs to the set of sets returned by $\mathbb{C}_{x,y}(\tau, i, [\![p]\!])$. See [46] for the others. The semantic postulates for operations [12] are:

$\tau \models_i \mathit{Create}(C(x,y,p,q))$ iff $\tau \not\models_i C(x,y,p,q)$ and $\tau \models_{i+1} C(x,y,p,q)$
$\tau \models_i \mathit{Discharge}(C(x,y,p,q))$ iff $\tau \models_i C(x,y,p,q)$ and $\tau \models_{[0,i+1]} q$
$\tau \models_i \mathit{Expire}(C(x,y,p,q))$ iff $\tau \models_i C(x,y,p,q)$ and $\tau_{[0,i+1]} \models \overline{p}$
$\tau \models_i \mathit{Violate}(C(x,y,p,q))$ iff $\tau \models_i C(x,y,p,q)$ and $\tau \models_{[0,i+1]} \overline{q}$
$\tau \models_i \mathit{Detach}(C(x,y,p,q))$ iff $\tau \models_i C(x,y,p,q)$ and $\tau \models_{[0,i+1]} p$

The notion of *residuation* allows tracking progress towards condition achievement in the real world. Following [46,61], the *residual* of a temporal expression q with respect to an event e, denoted as q/e, is the remainder temporal expression that would be left over when e occurs, and whose satisfaction would guarantee the satisfaction of the original temporal expression q. In this work we adopt exactly the same definition of residuation. In particular, temporal expressions not mentioning an event are independent of that event; conjoined or disjointed expressions can be treated modularly; and expressions can be incrementally progressed: a residuated expression embodies the relevant history, no additional history need be represented.

Residuation is used to compute commitment progress. Given a commitment $c = C(x,y,p,q)$ and an event e, c/e denotes $C(x,y,p/e,q/e)$. Commitment progression leads to one of the following outcomes: the commitment expires, it is violated, it is discharged, or its residual with respect to an event is computed (its antecedent and consequent are residuated). In the latter case, progression may be trivial in the sense that the commitment conditions are left unchanged. The following theorem [46], captures how a commitment progresses. Following [61], $\top$ means the temporal expression satisfied by every run and 0 an expression that is satisfied by no run.

Theorem 1 *If* $\tau \models_i C(x,y,p,q)$ *and* $\tau_{i+1} = e$, *then*

$$\begin{array}{ll}
\tau \models_{i+1} \mathit{Expire}(C(x,y,p,q)) & \textit{if } p/e \doteq 0 \\
\quad \mathit{Violate}(C(x,y,p,q)) & \textit{if } p/e \doteq \top,\ q/e \doteq 0 \\
\quad \mathit{Discharge}(C(x,y,p,q)) & \textit{if } q/e \doteq \top \\
\quad \mathit{Detach}(C(x,y,p,q)) & \textit{if } p/e \doteq \top \\
\quad C(x,y,p/e,q/e) & \textit{otherwise}
\end{array}$$

Intuitively, the theorem means that always the occurrence of an event e either leads an active commitments to a final state, or makes its antecedent and consequent conditions progress one step further.

3 Typing Agents through Social Relationships

Before explaining our typing system, it is useful to recall the characteristics of the multi-agent systems we are interested in, so as to make the challenges we have to face explicit.

First of all, we are interested in type checking *interacting* agents. In this context, a type checking system should, as hoped for in [71], be based on the agents' observable behaviors. In addition, the type checking should take into account the agents' ability to enact roles dynamically, as envisaged in [34]. This means that agents can start/stop playing roles at any time along their execution, and can even play more than one role at a time. Such a dynamicity poses a question about the legitimacy for an agent to play a role at a given time. This suggests that: (i) the fact that an agent starts playing a role must be dealt with as a special operation, that we call *enactment*; and (ii) the enactment of a role demands for some form of runtime *dynamic check*. Namely, the type checking must verify whether, at the time of the enactment, the candidate role player actually meets the expectations required by the role it wants to play. As noted above, in fact, agents can acquire behaviors at execution times, and these behaviors may allow them to play specific roles. Finally, we are interested in supporting the engineering of multi-agent systems so as to spread their application to real-world problems. This, however, demands for a typing system that helps a programmer in the development of the agents, especially as concerns their interactions with others. Thus, the typing system we aim at must be practical (i.e., easy to use), and capable of detecting interaction errors, such as the lack of some agents' behaviors that are necessary for the progression of a specific interaction. On the other hand, the typing system must not be overly prescriptive, by forcing an agent (or a programmer) to a course of actions for achieving a goal. In short, a strong requirement for our typing system is to be built upon a *declarative language* which, relying on the agent paradigm abstractions, be both simple to use and minimally prescriptive. In this section we show how this requirement can be met by relying on the precedence logic introduced in [61] for the composition of Web services.

3.1 Type checking for enactments

An agent can deliberate to enact a protocol role at any time of its execution. Thus, a type checking system should verify, dynamically, that only an agent adequately equipped to play the selected role is actually enabled to enact that role. In other words, the type checking should be a filter on the enactment function: the filter prevents an agent from playing a role unless it has a proper set of behaviors to carry out all the duties associated with the role to be enacted. Of course, the type checking is grounded on the assumption that the underlying protocol is "well-defined", i.e., each agent can satisfy any of its commitments without being forced to violate some other. In other words, the protocol must allow agents to achieve their own goals by coordinating with others so as to find a sequence of behavior executions that leads to the satisfaction of all the involved commitments. Before presenting the enactment checking, we thus characterize the class of *socially-progressive* protocols that support it.

Following [46], we say that an event e is relevant to $C(x,y,\ p,\ q)$ when e is involved either in p or q. That is, e is significant in the expiration, violation, discharge, detachment, or progression of the commitment. Definition 2 captures this fact.

Definition 2 An event $e \in \mathscr{B}$ is *relevant to* a commitment $C(x,\ y,\ p,\ q) \in \mathscr{C}$ iff one of the following holds: $p/e \not\equiv p$, $p/\overline{e} \not\equiv p$, $q/e \not\equiv q$, or , $q/\overline{e} \not\equiv q$.

Definition 3, brings into protocols the notion of "closeness" originally introduced for enactments in [12]. It states that in a closed protocol, for any relevant event, there is at least one role that can make it occur, and that all events generated by the protocol are relevant to some commitment generated by the same protocol.

Definition 3 A protocol P is *closed* iff for every commitment $c \in \mathscr{C}_P$, every event e that is relevant to c belongs to $\mathscr{B}_P$, and for each $e \in \mathscr{B}_P$ either there is a commitment $c \in \mathscr{C}_P$ such that e is relevant to c, or the event creates (cancels or otherwise explicitly manipulates) a commitment in $\mathscr{C}_P$.

Along the line of [45,44], we restrict to considering commitments whose debtor role has the powers to bring about the consequent condition, and whose creditor role has the powers to bring about the antecedent condition. This is specified by the following definition.

Definition 4 A protocol P is *role-distinct* iff for each commitment $C(x,y,p,q) \in \mathscr{C}_P$, the temporal expression p is made of events that amount to powers of y while the temporal expression q is made of events that amount to powers of x.[4] When needed, we use a subscript to denote the role having certain powers (e.g. p_y and q_x).

Let us denote by $\mathbf{e}$ a sequence $e_1, e_2, \ldots, e_n$ of events. Given two sequences $\mathbf{e}$ and $\mathbf{e}'$, we denote by $\mathbf{e}+\mathbf{e}'$ their concatenation. We also extend the notion of residual of a temporal expression q to a sequence of events $\mathbf{e}$ as follows: $q/\mathbf{e} = (\ldots((q/e_1)/e_2)/\ldots)/e_n$. If $q/\mathbf{e} \doteq \top$ and for all e_i in $\mathbf{e}$, $q/e_i \not\equiv q$ (all events are relevant to q), we say that the sequence $\mathbf{e}$ is an *actualization* of the temporal expression q. We denote by $\widehat{q}$ an actualization of q.

Finally, let $\mathbf{e}$ be the sequence $e_1, e_2, \ldots, e_n$, we denote by $\mathbf{e}_{[1,i]}$ the sequence $e_1, e_2, \ldots, e_i$, and by $\dot{\mathbf{e}}$ the temporal expression $e_1 \cdot e_2 \cdot \ldots \cdot e_n$ where all events are composed by means of the *before* operator.

Definition 5 Two temporal expressions q and u are:

- *mutually exclusive conditions*: if for all $\widehat{q}$ we have $u/\widehat{q} \doteq 0$ and for all $\widehat{u}$ we have $q/\widehat{u} \doteq 0$;
- *separable conditions*: if there exist $\widehat{q}$ and $\widehat{u}$ such that $q/\widehat{u} \equiv q$ and $u/\widehat{q} \equiv u$;
- *coordinable conditions*: if either q and u are separable conditions or there exist $\widehat{q} = \mathbf{e}_1 + \mathbf{e} + \mathbf{e}_2$ and $\widehat{u} = \mathbf{e}_3 + \mathbf{e} + \mathbf{e}_4$ such that $\dot{\mathbf{e}}_1$ and $\dot{\mathbf{e}}_3$ are coordinable conditions, and $\dot{\mathbf{e}}_2$ and $\dot{\mathbf{e}}_4$ are coordinable conditions.

Example 4 Some simple examples should clarify these notions. Let us consider the following expressions under the control of the same agent, omitted for brevity, $q = book \wedge confirm \wedge pay$ and $u = \overline{confirm}$; it is easy to see that q and u are mutually exclusive since the occurrence of $confirm$ would prevent the occurrence of any possible actualization for u, and vice versa. If we consider $u' = get\_address \wedge deliver$, we have now that q and u' are separable since they refer to different sets of events, and hence their actualizations can be interleaved freely. For instance, a possible sequence of event that can occur during execution would be $book \cdot get\_address \cdot confirm \cdot pay \cdot deliver$. Finally, if we consider $u'' = get\_address \wedge confirm \wedge deliver$ we have that q and u'' are coordinable. In fact, there exist actualizations $\widehat{q}$ and $\widehat{u''}$ for the two expressions that allow their progression to $\top$. For instance $\widehat{q} = book \cdot confirm \cdot pay$ and $\widehat{u''} = get\_address \cdot confirm \cdot deliver$ is a possible pair of actualizations that make q and u'' coordinable since $book$ and $get\_address$ are separable as well as pay and $deliver$. Another solution could be $\widehat{u''} = confirm \cdot get\_address \cdot deliver$, in fact $book$ is trivially separable with an empty event and pay is separable with the subsequence $get\_address \cdot deliver$. ■

Definition 6 (Coordinable commitments) Two commitments $c = C(x,y,p_y,q_x)$ and $c' = C(z,w,r_w,u_z)$ are *coordinable* when the following conditions hold:

- $x \not\equiv z$, $x \not\equiv w$, $y \not\equiv z$, and $y \not\equiv w$;

[4] In terms of [46], this implies that p is controlled by y and q is controlled by x.

- or $x \equiv z$ and either q_x and u_z are coordinable or p_y and r_w are mutually exclusive;
- or $x \equiv w$ and q_x and r_w are coordinable;
- or $y \equiv z$ and p_y and u_z are coordinable;
- or $y \equiv w$ and p_y and r_w are coordinable.

Namely, two commitments are coordinable if also their antecedent and consequent conditions are coordinable. The first condition asserts that when c and c' involve four different agents, then their temporal expressions are trivially coordinable as a consequence of the role-distinct assumption. The second condition considers the case in which the two commitments share the same debtor agent. In that case, c and c' are coordinable if their two consequent conditions are coordinable, or in case the antecedent conditions are mutually exclusive. In this second case, in fact, the debtor would be led to create only one of the two commitments. The third and fourth conditions, instead, consider situations in which an agent is both creditor for a commitment and debtor for the other. These situations demand that the expressions under the control of the same agent be coordinable. Finally, the last condition considers the case when both commitments share the same creditor agent. Also in this case it is required that the temporal expressions under the control of same agent be coordinable. Notably, the situations in which the two commitments are defined between the same two agents is not considered explicitly because already captured by a combination of the provided conditions. For instance, the combination of the second with the last conditions gives us the way to deal with the situation where the same agent is the creditor for c and c', and another agent is the debtor for the very same commitments.

Intuitively, a set of commitments is coordinable when there is a sequence of events, that allows the involved agents to make all the considered commitments progress until their discharge/detach (depending on whether each agent is debtor or creditor).

The previous definitions allow us to introduce the concept of *socially-progressive* protocol. Intuitively, a protocol is expected to be defined in such a way that it will allow the agents, playing its roles, to be in condition to discharge all the commitments that may be brought about in such role playing, i.e., to satisfy one such commitment without violating another.

Definition 7 (Socially-progressive protocol) A closed and role-distinct protocol P is *socially progressive* iff each pair of commitments c and c' in $\mathscr{C}_P$ is coordinable.

Let $\mathbf{cone}(\tau, i) = \{\tau' | \tau_{[0,i]} = \tau'_{[0,i]}\}$. Intuitively, the cone of a run at an index includes all possible future branches given the history (the part of the run up to the index) [46].

Proposition 1 *Given a socially progressive protocol P and a commitment $c = C(x, y, p_y, q_x)$ in $\mathscr{C}_P$, if $\tau \models_i Create(c)$, $\tau \models_j Detach(c)$, $i \leq j$, and $\tau \models_k Discharge(c)$, $j \leq k$, then for each $\widehat{p_y}$, and $\widehat{q_x}$, $\exists \tau' \in \mathbf{cone}(\tau, i)$ such that $\tau' \models_{i+n} Detach(c)$, where $\tau'_{[i+1,i+n]}$ amounts to $\widehat{p_y}$, and $\tau' \models_{i+n+m} Discharge(c)$, where $\tau'_{[i+n+1,i+n+m]}$ amounts to $\widehat{q_x}$.*

Proof Given an actualization $\widehat{p_y}$ and an actualization $\widehat{q_x}$, let us define τ' as the concatenation of $\tau_{[0,i]}$ with $\widehat{p_y}$, with $\widehat{q_x}$, and with any sequence of events not appearing previously. The run τ' exists because, on the one hand, since the protocol is *closed*, all the events that are relevant either to p_y or to q_x are generated inside the protocol. On the other hand, as a consequence of the role-distinct property, the events generated by any two agents can be interleaved freely. It follows that among the set of possible runs in $\mathbf{cone}(\tau, i)$ there exists a τ' where the first n events after the i-th step are generated by the actualization $\widehat{p_y}$, while the subsequent m events are generated by the actualization $\widehat{q_x}$. Now, by definition of $\widehat{p_y}$, we have that $\tau' \models_{i+n} Detach(c)$. Then, by definition of $\widehat{q_x}$, we have that $\tau' \models_{i+n+m} Discharge(c)$.

Notice that, the selected τ' may discharge a commitment and not another one the agent is also involved in as debtor, as it could cause the expiration of a commitment in which the agent is involved as creditor. In other words, we do not look for a τ' that necessarily satisfies all the commitments an agent is in charge of. The proof, in fact, just shows the existence of at least one τ' where the actualizations $\widehat{p_y}$ and $\widehat{q_x}$ do occur. However, this is not restrictive because since the protocol is socially progressive, any pair of such commitments is coordinable, thus there always exists a coordinable sequence of events that allows an agent to discharge/detach all.

An important consequence of Proposition 1 and Theorem 1 is that it is possible to program each agent's behavior as a reaction to a change in the social state. This is possible because each event occurrence causes the progression of at least one commitment even when commitment conditions are temporal expressions, that take more than one step to be satisfied or to fail. Thus, the significance of Property 1 is that, when a socially-progressive protocol is properly enacted by a group of agents, there always exists at least one course of events that brings all the commitments possibly created along the interaction to satisfaction. In other terms, the agents interacting via a socially-progressive protocol will never get stuck because no agent knows how, or has the capability, to make the interaction progress a step further. Through progression each such step is semantically captured in the social state. Of course, the key problem, now, is verifying that when an agent tries to enact a role in a socially-progressive protocol, that agent has the capabilities for accomplishing the role's duties. This is right the goal of our typing system, which aims at verifying that an agent implementation produces the proper subset of the relevant events in response to commitment progressions. Another consequence is that, since socially-progressive protocols are role-distinct, each agent can individually and locally check its capacity to play a protocol role – by being able to make the commitments, it may be involved into, progress to discharge/failure. Of course, having the capacity to satisfy a commitment does not mean that at run-time agents will never violate their commitments. Their autonomy in deliberating which actions to execute is not reduced in any way.

Definition 8 Given a socially-progressive protocol P, a commitment $c = C(x, y, p_y, q_x)$ in $\mathscr{C}_P$, the sequence of events $\mathbf{e}$, and an event e', such that $\mathbf{e}$ and $\mathbf{e} + e'$ are prefixes either of $\widehat{p_y}$ or of $\widehat{q_x}$, a *type expression* has the form $c/\mathbf{e} \rightarrow c/(\mathbf{e} + e')$.

Type expressions are used to specify those commitment progressions which are caused by the execution of behaviors – due to the fact that such behaviors make the event e' occur. In the following, we focus on behaviors that cause the occurrence of at most one event of a same protocol.[5]

Example 5 Let us consider the following commitment that we have already seen as part of the CNP protocol in Example 3: $c = Ci, p, propose, accept \vee reject$ Possible type expressions for c are, for instance $c/\{\} \rightarrow c/\{propose\}$ and $c/\{\} \rightarrow c/\{accept\}$. The first expression types a progression of c from a state where c is *conditional* to a state where it is *detached*. The second expression types a progression of c from *active* (i.e., conditional or detached) to *satisfied*.

As a more interesting example consider commitment $c' = C(mer, cus, book \cdot pay, procure \cdot pack \cdot (deliver \vee mail))$: merchant *mer* commits to customer *cus* to procure some required goods pack them and either deliver personally or send them via mail, provided that the customer books and pays for the goods. Possible type expressions for the merchant are $c'/\{\} \rightarrow c'/\{procure\}$; $c'/\{procure\} \rightarrow c'/\{procure, pack\}$ and $c'/\{procure, pack\} \rightarrow c'/\{procure,$

[5] This is not restrictive as it is possible to encapsulate many such behaviors into a new one.

pack, *deliver*$\}$. Only the last expression types a progression from state *detached* to state *satisfied*; whereas, the previous expressions are intermediate progressions which do not change the current state of c'. ■

Definition 9 (b-type) Given a socially-progressive protocol P and a set of commitments $c_1,\ldots,c_n$ in $\mathscr{C}_P$, a *b-type* is defined as a set of type expressions: $\{c_1/\mathbf{e}_1 \rightarrow c_1/(\mathbf{e}_1+e),\ldots,c_n/\mathbf{e}_n \rightarrow c_n/(\mathbf{e}_n+e)\}$, where the temporal expressions $\dot{\mathbf{e}}_1,\ldots,\dot{\mathbf{e}}_n$ are coordinable.

A b-type is used to type an agent behavior. For instance the merchant agent in Example 5 may be involved in, besides in commitment c', also in the following commitment $c'' = C(shipper, merchant, pack \cdot mail, ship)$. Note that c' and c'' are coordinable, and hence their progression must proceed synchronously at least for those shared events. Thus, the merchant may have a behavior *b-pack* whose *b-type* is $\{c'/\{procure\} \rightarrow c'/\{procure, pack\}, c''/\{\} \rightarrow \{pack\}\}$; meaning that *b-pack* generates the event *pack*, whose occurrence makes both c' and c'' progress. Notice that an agent behavior may have more than one b-type, each one referring to a different protocol. Indeed, a protocol represents the scope within which b-types can be defined and are meaningful. When a single behavior is adorned by more than one b-type, each within the scope of a different protocol, the behavior has an effect on each of such protocols. Namely, the events generated by the behavior execution make at least one commitment in each of these protocols progress.

Definition 10 Given a socially-progressive protocol P and a commitment c in $\mathscr{C}_P$, let b_1 and b_2 be two b-typed behaviors of some agent. We write $b_1 \sqsubseteq_c b_2$, if the b-type of b_1 contains $c/\mathbf{e} \rightarrow c/(\mathbf{e}+e')$ and the b-type of b_2 contains $c/(\mathbf{e}+e') \rightarrow c/((\mathbf{e}+e')+e'')$.

A sequence of behaviors $b_1,\ldots,b_n$ of an agent is an *ordered list* of behaviors *w.r.t.* c, if $b_i \sqsubseteq_c b_{i+1}$, $1 \leq i \leq n-1$. Let us denote such an ordered list by $[b_1,\ldots,b_n]_{\sqsubseteq_c}$. Any $[b_1,\ldots,b_n]_{\sqsubseteq_c}$ specifies a sequence $\mathbf{e}$ of events that makes the commitment c progresses along with the execution of the mentioned behaviors; more precisely, $\mathbf{e} = e_1,\ldots,e_n$, where b_i has b-type $c/\mathbf{e}_{[1,i-1]} \rightarrow c/(\mathbf{e}_{[1,i-1]}+e_i)$. We write $[b_1,\ldots,b_n]^{\mathbf{e}}_{\sqsubseteq_c}$ to make explicit that $\mathbf{e}$ is such a sequence of events.

Definition 11 Given a socially-progressive protocol P and a commitment $= C(x,y,p_y,q_x)$ in $\mathscr{C}_P$, a $[b_1,\ldots,b_n]^{\mathbf{e}}_{\sqsubseteq_c}$ is *complete* if $\mathbf{e}$ is an actualization of q_x or p_y.

Definition 12 (Debtor-compliance) Given a socially-progressive protocol P and a commitment $c = C(x,y,p_y,q_x)$ in $\mathscr{C}_P$, an agent a is *debtor-compliant* w.r.t. c if it has a set of behaviors $b_1,\ldots,b_n$ such that $[b_1,\ldots,b_n]^{\widehat{q_x}}_{\sqsubseteq_c}$.

Intuitively, an agent, that is debtor compliant w.r.t. a certain commitment, has a set of behaviors that, when executed in the proper order, will allow the agent to make the commitment progress from the Detached to the Discharged state. In other words, that agent is equipped with an implementation to deal with its obligations [36]. In order to enact a protocol role, an agent must be debtor-compliant with all the commitments where such a role appears as debtor.

Definition 13 (Creditor-compliance) Given a socially-progressive protocol P and a commitment $c = C(x,y,p_y,q_x)$ in $\mathscr{C}_P$, an agent a is *creditor-compliant* w.r.t. c if it has a set of behaviors $b_1,\ldots,b_n$ such that $[b_1,\ldots,b_n]^{\widehat{p_y}}_{\sqsubseteq_c}$, or such that $[b_1,\ldots,b_n]^{\widehat{\widehat{p_y}}}_{\sqsubseteq_c}$, or it has a behavior b whose b-type is $c/\mathbf{e} \rightarrow c/(\mathbf{e}+release_c)$, where $release_c$ is an event that causes the release of the commitment c.

Although we adopt the same perspective of Singh about social commitments, that an agent does not need to accept a commitment for which it is creditor nor it is committed to bringing about the antecedent condition, it is important is that the programmer is *aware* of the opportunity provided by the commitment, should one ever be created. A commitment that is not possible to detach nor to release by an agent is a symptom that the programmer may not have such awareness and for this reason it ought to be brought to the attention of the programmer. Consequently it needs to be tackled by the type checking as it may, indeed, amount to a programming mistake. For this reason, we demand, through Definition 13, that a creditor agent, even though not obliged to possess an actualization for the antecedent conditions of its commitments, has at least a behavior for releasing those commitments. Practically, a creditor agent must either be capable of bringing about the antecedent (making the commitment progress to Detached), or it must be capable of falsifying the antecedent (making the commitment progress to Expired), or finally it must be capable of releasing the commitment. In the last case, the idea is that a creditor agent should not be forced to possess implementations concerning commitments to which it is not interested. Thus, for completeness reasons, the enactment checking verifies whether an agent has considered all the commitments in which it can appear as creditor. This is solved transparently by checking that the agent can detach or (not exclusive) can release each of its commitments.

Proposition 2 *Given a socially-progressive protocol P and a commitment $c = C(x, y, p_y, q_x)$ in $\mathscr{C}_P$, let a be an agent that enacts role x and that is debtor-compliant w.r.t. c. If $\tau \models_i Detach(c)$ and $\tau \models_j Discharge(c)$, $i \leq j$, then a has an implementation $[b_1, \ldots, b_n]^{\widehat{q_x}}_{\sqsubseteq_c}$, and $\exists \tau' \in \textbf{\textit{cone}}(\tau, i)$ such that $\tau' \models_{i+n} Discharge(c)$, where $\tau'_{[i+1,i+n]}$ amounts exactly to the same actualization $\widehat{q_x}$ of agent a's implementation.*

Proof Since the agent a is debtor compliant w.r.t. the commitment c, it has a complete implementation $[b_1, \ldots, b_n]^{\mathbf{e}}_{\sqsubseteq_c}$, where $\mathbf{e}$ is an actualization of q_x. By Proposition 1 and by definition of actualization, for each run τ', such that the sequence of events $\tau'_{[i+1,i+n]}$ is an actualization $\widehat{q_x}$, we have that $\tau' \models_{i+n} Discharge(c)$. In particular this will hold also for the actualization $\mathbf{e}$. Moreover, since the protocol is role-distinct, agent a is independent of the other agents in bringing about q_x, by executing $[b_1, \ldots, b_n]^{\mathbf{e}}_{\sqsubseteq_c}$.

Proposition 3 *Given a socially-progressive protocol P and a commitment $c = C(x, y, p_y, q_x)$ in $\mathscr{C}_P$, let a be an agent that enacts role y and that is creditor-compliant w.r.t. c. If $\tau \models_i Create(c)$ and $\tau \models_j Detach(c)$, $i \leq j$, then a has an implementation $[b_1, \ldots, b_n]^{\widehat{p_y}}_{\sqsubseteq_c}$ and $\exists \tau' \in \textbf{\textit{cone}}(\tau, i)$ such that $\tau' \models_{i+n} Detach(c)$, where $\tau'_{[i+1,i+n]}$ amounts to $\widehat{p_y}$, or it has a behavior b whose b-type is $c/\mathbf{e} \rightarrow c/(\mathbf{e} + release_c)$.*

Proof The proof follows an schema that is analogous to the one of Proposition 2.

Propositions 2 and 3 mean that a group of agents can enact a role in a socially-progressive protocol only when they are creditor and debtor compliant for each commitment in which their role appears either as creditor or as debtor. When this happens, Property 1 is preserved: the agents enacting the protocol can produce at least one course of events that brings to the satisfaction of any commitment possibly created during the interaction. This excludes typical interactions errors, such as, deadlocks, situations where no agent knows how to proceed, or where an agent is forced to violate one of its commitments for satisfying another one.

In fact, an agent might be involved in many commitments of a same protocol at a time, either as debtor or as creditor, and, in general, the discharge of a commitment may cause the violation (or the expiration) or another.

However, we focus on socially-progressive protocols, which guarantee that each pair of commitments is coordinable (i.e., there is a way to satisfy both). On the other hand, b-types too require the involved temporal expressions to be coordinable. This means that each agent has the possibility to generate a coordinable sequence of events that allows it to discharge/detach all of its commitments.

Of course, it could be interesting to consider also other commitment state transitions in the typing, besides those yielded by the events create, detach, and discharge occurring in this order. We decided to focus on this kind of transitions because they are the ones that bring about opportunities and obligations: creditor-compliance aims at guaranteeing that agents be aware of the possible opportunities, while debtor-compliance guarantees that agents are endowed with behaviors that allow them to face their possible obligations. Opportunities are given in terms of commitments towards agents, while obligations are commitments taken by agents. We conclude this section with some examples to summarize the concepts at the basis of our type checking, and to show how the type checking actively controls the agents' role enactments.

Example 6 Let us consider again Example 3, and let us show that this protocol is socially progressive, in fact: (1) each relevant event is generated by some role in the protocol, and each generated event is relevant to at least one of the commitments that can possibly be created along the protocol execution (*closeness*); (2) each event can only be generated by one role (*role distinctness*); and (3) the commitments are coordinable. To verify this it is sufficient to note that, during the execution of the protocol, two commitments can be generated: $c_1 = C(i, p, propose, accept \vee reject)$ and $c_2 = C(p, i, accept, done \vee failure)$. By Definition 6, the two commitments are coordinable when the temporal expressions *propose* and $done \vee failure$ are coordinable, and when $accept \vee reject$ and *accept* are coordinable. The first pair of expressions are separable, and hence coordinable. For the second pair, instead, there exists an actualization, consisting of just the event *accept*, that coordinates the two expressions. Observe that if we use the representation of CNP described in Example 1, the involved temporal expressions (i.e., $accept \vee \overline{accept}$ and *accept*) would not be separable. Still they would remain coordinable (because of the common actualization *accept*). For the sake of completeness, had the two temporal expressions been *accept* and $\overline{accept}$, they would not have been coordinable and the protocol would not have been socially progressive. Assume now that an agent is willing to play the *initiator* role, what behaviors should the agent possess in order to pass the enactment checking? (In the following we name the behaviors following the convention *bvr-event*, meaning that the execution of the behavior body will cause the occurrence of the social event *event*.) Let us assume that it is equipped with the following set of b-typed behaviors:

- *bvr-accept*: $\langle c_1/\{\} \rightarrow c_1/\{accept\}\rangle$ the type denotes that after the execution of *bvr-accept*, commitment c_1 will progress (to discharge) with the occurrence of event *accept*.
- *bvr-reject*: $\langle c_1/\{\} \rightarrow c_1/\{reject\}, c_2/\{\} \rightarrow \{release_{c_2}\}\rangle$, here the type highlights that the behavior impacts on two commitments: c_1 progresses (to discharge) with the event *reject*, whereas commitment c_2 is released.

Thanks to these behaviors, the agent will pass the enactment checking. In fact, the checking verifies whether the agent is both debtor-compliant w.r.t. each commitment in the protocol where *initiator* appears as debtor, and creditor-compliant w.r.t. each commitment where *initiator* appears as creditor. The first requirement is satisfied since the agent possesses at least one actualization for the consequent condition in c_1 (the only commitment where the initiator is debtor). Indeed, the agent has two alternative actualizations to be debtor compliant:

$[brv\text{-}accept]^{\mathbf{accept}}_{c_1}$ and $[brv\text{-}reject]^{\mathbf{reject}}_{c_1}$. The second requirement, to be creditor-compliant for commitment c_2, is satisfied again by behavior *bvr-reject*, which represents the actualization $[brv\text{-}reject]^{\mathbf{release}}_{c_2}$, admissible for the creditor role. Analogously, when an agent intends to play the *participant* role, a possible set of behaviors that satisfy the enactment checking is:

- *bvr-done*: $\langle c_2/\{\} \rightarrow c_2/\{done\}\rangle$ which makes commitment c_2 progress to discharge;
- *bvr-refuse*: $\langle c_1/\{\} \rightarrow c_1/\{release_{c_1}\}\rangle$ whose effect is releasing commitment c.

The first behavior makes the agent debtor-compliant, the second creditor-compliant. ■

Example 7 Consider the trade-like interaction in Table 3 where a merchant *m* sells goods to a purchaser *p*. Intuitively, a merchant commits to a purchaser to procure (if not already

Table 3: A buying and selling protocol definition.

ROLES	POWERS	SOCIAL EFFECTS
MERCHANT	*offer*	create(*C(m, p, book, procure · pack)*)
	sell	create(*C(m, p, pay, pack · deliver)*)
	procure	–
	pack	–
	deliver	–
PURCHASER	*book*	create(*C(p, m, ⊤, pay)*)
	pay	–

available), pack and deliver goods the purchaser has payed for (*offer* and *sell* actions). On the other side, the purchaser commits to the merchant that it will pay for the goods it has booked (*book* action). Thus, the evolution of the protocol can generate the following commitments: $c_1 = C(m, p, pay, pack \cdot deliver)$, $c_2 = C(m, p, book, procure \cdot pack)$, and $c_3 = C(p, m, \top, pay)$. The protocol is socially progressive since it is closed, role distinct and coordinable. Let us now assume that an agent *a* is willing to play role *merchant*. Agent *a* has to take care of two commitments where *merchant* appears as debtor (i.e., c_1 and c_2), and one commitment where it occurs as creditor (i.e., c_3). On the debtor side, event *pack* is mentioned in the consequent conditions of both c_1 and c_2; therefore, the implementations of their actualizations must be coordinated; let us show how this is done by considering the following set of behaviors:

- *bvr-procure*: $\langle c_1/\{\} \rightarrow c_1/\{procure\}\rangle$;
- *bvr-pack*: $\langle c_1/\{procure\} \rightarrow c_1/\{procure, pack\}, c_2/\{\} \rightarrow c_2/\{pack\}\rangle$;
- *bvr-deliver*: $\langle c_2/\{pack\} \rightarrow c_2/\{pack, deliver\}\rangle$.

These behaviors realize two actualizations for the two commitments c_1 and c_2. In fact, commitment c_1 is discharged via the ordered sequence of behaviors $[bvr\text{-}procure, bvr\text{-}pack]^{\mathbf{e}}_{c_1}$, where $\mathbf{e}$ equals $\{procure, pack\}$. Commitment c_2 is discharged by $[bvr\text{-}pack, bvr\text{-}deliver]^{\mathbf{e}'}_{c_2}$, where $\mathbf{e}' = \{pack, deliver\}$. The two actualizations represent a coordinated way to satisfy both c_1 and c_2, and hence the agent is debtor-compliant w.r.t. all its commitments. In fact, if a purchaser booked some goods and immediately after payed for it, agent *a* would be committed to satisfy both commitments; however, thanks to the coordination of the two actualizations, the consequent condition of c_1 will be pursued only after the achievement of event *pack*; i.e., only after the goods are actually at the merchant's disposal. On the creditor side, the merchant must be equipped with a behavior for releasing commitment c_3 in order to be creditor-compliant; i.e., *bvr-release*: $\langle c_3/\{\} \rightarrow c_3/\{release_{c_3}\}\rangle$. ■

Example 8 Let us consider a protocol role that requires to its players to tackle a commitment, whose consequent is $(pack \cdot mail) \vee (give\_sh \cdot send\_tn)$, meaning that that agent should either pack some item and then mail it through the post office, or give it to a shipper, and then send the tracking number to the client. Roughly, debtor-compliance checks that an agent willing to enact the role declares to have some behaviors for tackling this commitment. It also checks that such behaviors produce an actualization of the temporal expression. An agent, declaring to possess the abilities to tackle the commitment, passes the check if it is able to perform at least one of the two encoded transport procedures, but it would not pass the check if, for instance, it had behaviors for *pack* and *give_sh* only, which would allow it to pack the item and give it to a shipper for delivery. These two behaviors together, in fact, would not constitute a complete actualization for the consequent condition of c_2. ■

4 Implementing the Enactment Type Checking in 2COMM

The typing system discussed in the previous sections is independent of any specific multiagent platform, it only requires the notion of commitment to be native in it. In order to show how the type checking can be made concrete, we present a possible implementation that is grounded upon an existing multiagent platform supporting commitments, and show how b-types can be added to it and used. The starting point is 2COMM ([6,9,10]), a middleware that, integrated with existing multiagent platforms, realizes an agent programming environment where social relationships, modeled as commitments, are made available as first-class programming elements. So far, two bridges were developed for 2COMM: 2COMM4JADE allows to use commitments with *JADE* [18] agents, while 2COMM4JACAMO allows *JaCaMo* [22] agents to manipulate commitments. For the sake of exposition, we only discuss here the implementation for 2COMM4JADE. Before getting into details of the implementation of the typing system, we briefly overview the 2COMM architecture.

4.1 2COMM Quick Tour

Figure 2 sketches an excerpt of 2COMM's architecture (only the most relevant classes are shown). In 2COMM commitment-based interaction protocols are realized by means of CArtAgO [52] artifacts. Specifically, the core of 2COMM extends CArtAgO artifacts by providing the means for the management, the maintenance, and the update of the social state, that is associated with each instance of a protocol artifact. The 2COMM protocol artifact exhibits roles that specify how agents can manipulate the social state; agents playing roles acquire the *powers* for creating and affecting commitments.

2COMM is organized in layers as follows. At the top of the hierarchy, the `CommunicationArtifact` class provides agents with a basic tool for mediated communication that extends the CArtAgO tuple space with the additional notion of *roles* (see `CARole` class). Thus, an agent intending to use an instance of `CommunicationArtifact` has to play a role therein defined. To this aim, `CommunicationArtifact` defines two operations, `enact` and `deact`, that are used at runtime by agents when they access or release an instance of such a class. Notably, `enact` already implements some basic form of role enactment checking, but it is limited to syntactic issues (e.g., role name correctness). Intuitively, a natural way to implement the type checking is to properly extend `CommunicationArtifact`, and the `enact` method, in particular.

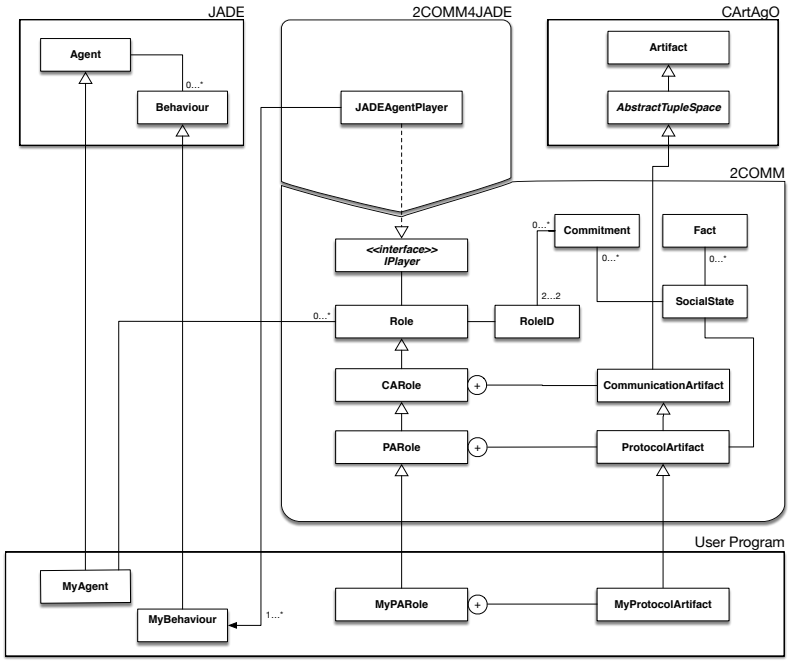

Fig. 2: 2COMM and its use in programming agents and protocols.

At the next level, the `ProtocolArtifact` class extends `CommunicationArtifact` by adding the notions of commitment and social state, and by providing the needed operations to manipulate commitments (i.e., create, discharge, cancel, release, assign, delegate). Note that `PARole` extends, via `CARole`, the class `Role`, which in turn extends the CArtAgO class `Agent`, and contains the basic manipulation logic of CArtAgO artifacts. Since roles extend this super-type, their instances will be able to perform operations on artifacts, whenever the role players will decide to do so. `Role` provides both static methods for creating artifacts and basic operations that are used during role enactment/deactment. In particular, on role enactment, an object of class `RoleId` is created. This is a unique identifier of the created role instance within the corresponding instance of `ProtocolArtifact`.

Class `SocialState` is the repository of social events that occur during an interaction. It contains instances of the classes `Commitment` and `SocialFact` (i.e., events that represent occurrences of protocol actions). The integrity and coherence of the social state is maintained by this class, which also provides a collection of methods for querying the state of the interaction.

The connection between JADE and 2COMM is realized through the 2COMM4JADE connector, which basically contains the class `JadeAgentPlayer`. This is an implementation of the interface `IPlayer`. Its instances represent JADE agents that can use a protocol by playing its roles.

Since `CommunicationArtifact` and `ProtocolArtifact` are abstract classes, they cannot be instantiated. A programmer using 2COMM4JADE for programming the interaction among some agents, needs just to extend `ProtocolArtifact`, and to exploit operations on commitments as primitives for the specification of the protocol actions (i.e. to implement their social meaning, see for instance `MyProtocolArtifact`). The

roles of a programmer-specified protocol are realized as inner classes of the class that extends `ProtocolArtifact` (see, for instance, `MyPARole` in the figure). This solution is adopted from [15] and, in particular, it realizes the definitional dependence principle (a role is meaningful only within the institution it belongs to). Such classes extend the abstract class `PARole` that is, in turn, an inner class of `ProtocolArtifact`. It provides both the primitives for *using the social state* (e.g., for asking in which commitments 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 may query if the social state contains a commitment with a specific condition as consequent, via the method `existsCommitmentWithConsequent(InteractionStateElement el)`. Alternatively, an agent may implement the inner interface `ProtocolObserver` (not reported in the figure), and thus be notified about the occurrence of specific social events. The operation `deact` is, instead, used by an agent to stop playing a role. Note that a protocol instance keeps track of which agent is playing what role by using the property `enactedRoles`.

4.2 Extending 2COMM with the Typing System

Now that we have sketched the general architecture of 2COMM, we can proceed with the discussion of how the typing system can be implemented upon it. Recall that our approach basically relies on two aspects: the *requirements* that are associated with each role in a given protocol, and the *capabilities* possessed by an agent at the time of the enactment of a role. Role requirements coincide with the commitments in which the given role can be involved, whereas capabilities are traced by means of *b-types*. Thus, these are the concepts that the implementation has to make concrete. Moreover, recall that the type checking we aim at must be *dynamic*: it must be performed at runtime just when an agent tries to enact a role. The type checking has to verify whether the agent's implementation is such to make the agent creditor- and debtor-compliant for each commitment involving the enacted role.

A natural way to approach the problem is to extend the `ProtocolArtifact` class. This is done by `TypedProtocolArtifacts`, which adds the type-checking mechanisms to a "standard" 2COMM `ProtocolArtifact`. Intuitively, `TypedProtocolArtifacts` overrides the basic enactment checking in `CommunicationArtifact`: when an agent invokes `enact` on a `TypedProtocolArtifact` instance for a given role, the operation checks the requirements associated with that role against the agent's type (i.e., the set of behaviors that are exhibited by the agent through the enact operation). We will describe deeply this operation in the following.

Before that, we need to present how the notions of role requirements and b-type are implemented. In order to be conservative with the existing 2COMM middleware, and to add the typing system as an additional feature that a programmer can use if she wants to, we have based the implementation of role requirements and b-types on *Java annotations* [66]. These are commonly used to provide meta-data about program elements (methods, classes, and fields), which can be used to perform activities on program code, for example by developing tools and code generation mechanisms. Relevant meta-data for our typing system are specified for *class* elements (`@Target({ElementType.TYPE})`); they must be recorded in the class files and retained at runtime (`@Retention({RetentionPolicy.RUNTIME})`).

The basic idea is that, thanks to the annotation mechanism, a programmer can create a bridge between a protocol role and the class defining its requirements, and between an agent behavior and its b-type. Let us start with the bridge between roles and their requirements.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface RoleType {
  Class<? extends RoleRequirements> requirements();
  int interactionCardinality() default 1;
}
```

Listing 1: The annotation `RoleType`.

Listing 1 reports the implementation of the `RoleType` annotation. According to the notions of debtor- and creditor-compliant (see definitions 12 and 13, respectively), this annotation allows specifying a set of commitments (i.e., `requirements`) that a role player should be able to fulfill.

The annotation also contains the property `interactionCardinality`, that specifies whether a role can be concurrently played by many agents – as it is, for instance, the case of the CNP role *participant*. When implementing a protocol role, it is necessary to mark the corresponding (inner) class with this annotation, specifying also the class which implements the desired requirements. That class must extend `RoleRequirements` (reported in Listing 2).

```
public abstract class RoleRequirements {
  final private Set<Commitment> C_DEFINITION;
  final private Set<Commitment> D_DEFINITION;
  final private String ROLE_NAME;
  protected RoleRequirements(String roleName,
      Commitment[] commitsDDefinition, Commitment[] commitsCDefinition) {
    this.ROLE_NAME = roleName;
    D_DEFINITION = new HashSet<Commitment>();
    for (Commitment c : commitsDDefinition) D_DEFINITION.add(c);
    C_DEFINITION = new HashSet<Commitment>();
    for (Commitment c : commitsCDefinition) C_DEFINITION.add(c);
  }
  private boolean checkCommitmentSatisfied(Commitment c,
      ArrayList<BType> bTypes) {
    TreeNode root = new TreeNode(new PartialActualization());
    ArrayList<TreeNode> leafNodes = new ArrayList<TreeNode>();
    updateTree(root, bTypes, leafNodes);
    LogicalExpression le = (LogicalExpression)c.getConsequent();
    for (TreeNode t : leafNodes) {
      for (ProtocolAction pa : t.partialAct.getActionList())
        update(le, pa);
      if (le.isVerified()) return true;
      le.resetVerifiedToFalse();
    }
    return false;
  }
  public boolean isEnactable(ArrayList<BType> bTypes) {
    boolean debt-ok = true, cred-ok = true;
    ArrayList<BType> arrNew;
    HashMap<Commitment,ArrayList<BType>> mapCommitTypes =
      new HashMap<Commitment,ArrayList<BType>>();
    for (BType bt : bTypes) {
      for (Commitment c : bt.getCommitments()) {
        if (mapCommitTypes.containsKey(c))
          mapCommitTypes.get(c).add(bt);
```

```
36          else {
37            arrNew = new ArrayList<BType>();
38            arrNew.add(bt);
39            mapCommitTypes.put(c, arrNew);
40          }
41        }
42      }
43      // CHECKING CREDITOR COMPLIANCE
44      if (!C_DEFINITION.isEmpty()) {
45        for (Commitment c : C_DEFINITION) {
46          if (!mapCommitTypes.containsKey(c)) return false;
47          cred-ok &= checkCommitmentDetachment(c, mapCommitTypes.get(c));
48        }
49      }
50      // CHECKING DEBTOR COMPLIANCE
51      for (Commitment c : D_DEFINITION) {
52        if (!(mapCommitTypes.containsKey(c))) return false;
53        debt-ok &= checkCommitmentSatisfied(c,
54          mapCommitTypes.get(c));
55      }
56      return debt-ok && cred-ok;
57    }
58  }
```

Listing 2: The `RoleRequirements` class.

`RoleRequirements` has three fields: `C_DEFINITION` and `D_DEFINITION` are commitment sets, and hence actually encode the role requirements; `ROLE_NAME` is the name of the role under consideration. Intuitively, actual role types constructors will invoke the superconstructor and specify proper arrays of commitments, that must be used to verify debtor compliance and creditor compliance respectively. `RoleRequirements` provides the method (`isEnactable`, line 27) to check if a role with specific requirements can be enacted by an agent with specific behavior types. This method is invoked by method *enact* of `TypedProtocolArtifact`.

For each commitment that is to be checked when verifying debtor compliance, the method `checkCommitmentSatisfied` (line 53) is called. This method (lines 13 – 26) builds a tree-like structure in order to verify whether the b-types, provided for the enactment, compose the required actualization (i.e., they can be used to make the commitment progress to satisfaction). *Creditor compliance* is checked in a similar way. The only difference is that in this case it is not required to have a complete actualization because of the possibility to rely on the *release* operation.

To give an understanding of how role requirements are practically implemented, let us quickly show how the role *merchant* in Example 7 can be defined in terms of its requirements. By construction, the protocol has to provide a clear definition of debtor-compliance to each role in terms of commitments. For the merchant role, the commitments to be considered are $c_1 = C(i, p, propose, accept \vee reject)$ and $c_2 = C(p, i, accept, done \vee failure)$. The *merchant*'s requirements are therefore implemented in class `MerchantRequirements` in Listing 3.

```
1 public class MerchantRequirements extends RoleRequirements {
2   public MerchantRequirements() {
3     super("Merchant",
4       new Commitment[]{
5       new Commitment(SellingProtocol.MERCHANT_ROLE,
6         SellingProtocol.PURCHASER_ROLE, "book",
7         new CompositeExpression(LogicalOperatorType.THEN,
```

```
8             new Fact("procure"), new Fact("pack"))),
9         new Commitment(SellingProtocol.MERCHANT_ROLE,
10          SellingProtocol.PURCHASER_ROLE, "pay",
11          new CompositeExpression(LogicalOperatorType.THEN,
12          new Fact("pack"), new Fact("deliver")))
13        },
14        null); // no commitments required for creditor compliance
15          }
16  }
```

Listing 3: Role requirements for merchant.

Since commitments are native in 2COMM, the role requirements implementation is quite simple: the developer defines the new requirement-class `MerchantRequirements` by extending the superclass `RoleRequirements`. In the constructor of the new class, the supercostructor is invoked by providing a name for the role, an array of commitments for debtor-compliance, and an optional array of commitments for creditor-compliance (in this example the latter is null).

Let us now consider the agent side and present how b-types are implemented. Following Definition 9, the type of a behavior B can be thought of as a tuple $\langle$*Commitment, Partial-Actualization, Protocol-Action*$\rangle$, where:

1. *Commitment* is the particular commitment that the behavior B manipulates;
2. *Partial-Actualization* is a list of protocol actions. It specifies which actions are assumed to have already been executed by the agent when B is executed;
3. *Protocol-Action* is the (only) action of the protocol the agent is going to perform by executing the behaviour B.

Annotation `BehaviourType`, similar to `RoleType`, is used to specify the b-type of a behavior, i.e. which commitments a behavior is capable to make progress.

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ElementType.TYPE})
3 public @interface BehaviourType {
4   Class<? extends BType> bType();
5 }
```

Listing 4: The annotation `BehaviourType`.

Indeed, the actual b-type of an agent's behavior must be an extension of class `BType`, whose implementation is reported in Listing 5.

```
1  public abstract class BType {
2    private Commitment[] commitments;
3    private PartialActualization partialActualization
4                          = new PartialActualization(this);
5    private ProtocolAction newAction;
6    public BType(Commitment[] c, PartialActualization pAct,
7                                       ProtocolAction newAct) {
8      commitments = c;
9      partialActualization = pAct;
10     newAction = newAct;
11   }
12   public BType(BType previousBType, ProtocolAction newAct) {
13     this.commitment = previousBType.getCommitment();
14     this.partialActualization = previousBType.getPartialActualization()
15                                                 .clone();
16     this.partialActualization.addAction(previousBType.getNewAction());
17     this.newAction = newAct;
```

```
18    }
19    // *** getters and setters... ***
20  }
```

Listing 5: The `BType` basic class.

Let us give a quick example of how b-types are actually implemented on the agent side taking again under consideration Example 7, and assuming that we want to implement an agent that will enact the role *merchant*. The agent will be equipped with a set of behaviors, each of which must be bound to a b-type by means of the annotation mechanism. For instance, the agent will have behavior `PackBehaviour`, that is intended to generate the (social) event *pack* by invoking the namesake protocol action. The type for such a behavior is defined by extending the `BType class`; Listing 6 shows its implementation.

```
1  public class BvtPackBType extends BType {
2    public BvtPackBType() {
3      super(new Commitment[]{
4        new Commitment(SellingProtocol.MERCHANT_ROLE,
5          SellingProtocol.PURCHASER_ROLE, "book",
6          new CompositeExpression(LogicalOperatorType.THEN,
7            new Fact("procure"), new Fact("pack"))),
8        new Commitment(SellingProtocol.MERCHANT_ROLE,
9          SellingProtocol.PURCHASER_ROLE, "pay",
10         new CompositeExpression(LogicalOperatorType.THEN,
11         new Fact("pack"), new Fact("deliver")))
12       },
13       new ProtocolAction("pack"));
14         }
15 }
```

Listing 6: BType of `bvt-pack` behaviour.

Thus, `BType` specifies which commitments progress as a consequence of the execution of a particular `ProtocolAction`. Note that, even though a behavior involves only one protocol action, it may cause the progression of many commitments. In the above example, the execution of *pack* causes the progression of both c_1 and c_2.

Finally, `BvtPackBType` must be linked to a specific behavior, `PackBehaviour` in our case. This is accomplished as in Listing 7 by means of the Java annotation mechanism.

```
1  ... agent code ...
2    @BehaviourType(bType = BvtPackBType.class)
3    public class PackBehaviour  {
4      ... behaviour specific logic ...
5    }
```

Listing 7: Linking a BType to an agent behavior.

`RoleRequirements` and `BType` abstract classes, together with `@RoleAnnotation` and `@BehaviourType` annotation classes realize the infrastructure upon which the type checking relies. The actual verification, however, is performed at runtime when an agent tries to play a specific role of a protocol extending `TypedProtocolArtifact`. As noted above, `TypedProtocolArtifact` inherits from `CommunicationArtifact` method `enact`, which in turn invokes method `checkRoleRequirements`. However, while `checkRoleRequirements` in `CommunicationArtifact` just makes syntactic name controls, in `TypedProtocolArtifact` this method is overridden to implement the comparison between the role requirements and b-types. This is the core of the type checking system and its implementation is reported in Listing 8.

```
1  protected boolean checkRoleRequirements(String roleName, IPlayer player) {
2    if (super.checkRoleRequirements(roleName, player)) {
3      JadeBehaviourPlayer jPlayer = (JadeBehaviourPlayer) player;
4      Behaviour[] offeredPlayerBehaviours = jPlayer.getJadeBehaviour();
5
6      Class<? extends Behaviour> behClass;
7      ArrayList<Annotation> behaviourTypeAnnotations
8            = new ArrayList<Annotation>();
9      Annotation behaviourTypeAnnotation;
10     for (Behaviour beh : offeredPlayerBehaviours) {
11       behClass = beh.getClass();
12       behaviourTypeAnnotation
13       = behClass.getAnnotation(BehaviourType.class);
14     if (behaviourTypeAnnotation == null) return false;
15     behaviourTypeAnnotations.add(behaviourTypeAnnotation);
16     }
17     String roleClassName = (this.getClass().getName())
18       + " dollarsign " + roleName;
19     Class<?> roleClass = Class.forName(roleClassName);
20     Annotation roleAnnotation =
21       roleClass.getAnnotation(RoleType.class);
22     if (roleAnnotation == null) return false;
23     RoleRequirements roleReq = ((RoleType)roleAnnotation)
24        .requirements().getDeclaredConstructor().newInstance();
25     ArrayList<BType> overallBTypes
26        = mergeBTypes(behaviourTypeAnnotations);
27     return rolereq
28        .isEnactable(overallBTypes);
29   } else return false;
30 }
```

Listing 8: The method `checkRoleRequirements` with commitment-based type-checking mechanisms.

`checkRoleRequirements` performs an initial check on the required role name– that must be included in the definition of the protocol– by invoking the namesake method of the superclass (line 2). The behaviors offered for enactment are parsed (line 10) and all `@BehaviourType` annotations are extracted. Analogously, the `@RoleType` annotation is retrieved from the definition of the role (line 17). Now, requirements can be taken from the `requirements()` method of the annotation (line 23); behavior types are collected (line 26), and both requirements and behavior types are used to check if the agent can enact the role (line 27).

5 Example: Adding B-Types and Requirements to a CNP Implementation

For the sake of completeness, we outline how the well-known Contract-Net Protocol modeled in terms of a commitment-based protocol in Example 2 can be typed with our methodology. An implementation in 2COMM4JADE of such a protocol has been discussed in [6]. In this section, we show how the typing system can be added on top of such an implementation. Recall that CNP encompasses two roles: *initiator* and *participant*, and each of them is associated with some "powers", having a meaning in terms of commitment operations. Figure 3 outlines the overall architecture for the typed CNP. The picture highlights the classes (and their mutual relations) that are part of the `typing` package, which extends 2COMM, and the classes that have to be implemented for adding the type checking feature to the CNP implementation. For the sake of readability, we abstract with `InitiatorBehaviour` the

set of behaviors `InitiatorAgent` is equipped with. As discussed in Example 6, these are *bvr-cfp*, *bvr-accept*, and *bvr-reject*; namely, one for each role's power. In the same way, we abstract with `TypeInitiator` the behavior type that is associated with each agent's behavior. Note that we use a non-standard notation to represent such an association since the relation between a behavior and its type is actualized via the Java annotation mechanism (as we will show below). A similar consideration holds for the *participant* role.

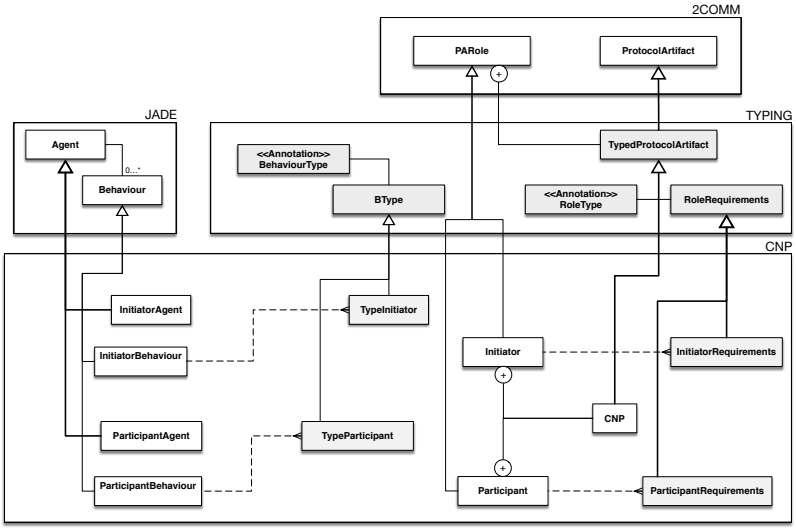

Fig. 3: Excerpt of the typing system architecture, and the example of the typed CNP.

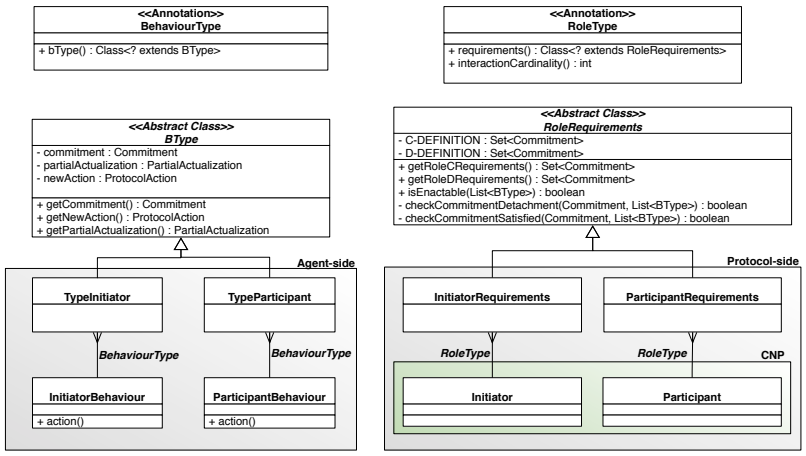

Fig. 4: UML excerpt of 2COMM typing framework.

Figure 4 details the relations existing between the classes of the `typing` package, and those in the CNP implementation. In particular, on the protocol side, two requirement classes are specified, for the *initiator* and *participant* roles, respectively. Each requirement class specifies what commitments an agent playing that role must be debtor/-creditor compliant. Notably, these requirement classes are related, via the Java annotation mechanism, to the original `Initiator` and `Participant` classes of the CNP implementation presented in [6]. This points out how the typing checking is a feature that can be added to already existing code with very little effort. On the agent side, two implementations of `BType` are showed, again to ease the readability of the picture, one for typing *initiator* agents and the other for typing *participant* agents. `CNPArtifact`, Listing 9, is a slight update of the original class discussed in [6]; changes regard: 1) the superclass now is `TypedProtocolArtifact`, and 2) the annotations that create a bind between each role and the class implementing its requirements. In Listing 9 this can be seen in line 3 for the *initiator* role, where the annotation for the nested class `Initiator` is used to create a reference with class `InitiatorRequirements`, and also in line 10 where a similar annotation is set for the *participant* role.

```
1  public class CNPArtifact extends TypedProtocolArtifact {
2    // ...
3    @RoleType(requirements = InitiatorRequirements.class)
4    public class Initiator extends PARole {
5      public Initiator(IPlayer player) {
6        super(INITIATOR_ROLE, player);
7      }
8      // ...
9    }
10   @RoleType(requirements = ParticipantRequirements.class)
11   public class Pariticipant extends PARole {
12     public Participant(IPlayer player) {
13       super(PARTICIPANT_ROLE, player);
14     }
15     // ...
16   }
17 }
```

Listing 9: Definition of `Initiator` role.

Class `InitiatorRequirements` defines the requirements for role *initiator*. Role requirements simply amount to the list of commitments involving the given role either as a debtor or creditor, to which a role playing agent will have to comply with:

```
1  public class InitiatorRequirements extends RoleRequirements {
2    public InitiatorRequirements() {
3      super("Initiator",
4      new Commitment[]{
5        new Commitment(CNPArtifact.INITIATOR_ROLE, CNPArtifact.PARTICIPANT_ROLE,
6          new Fact("propose"),
7          new CompositeExpression(LogicalOperatorType.OR,
8              new Fact("accept"), new Fact("reject")))
9      },
10     new Commitment[]{
11       new Commitment(CNPArtifact.PARTICIPANT_ROLE, CNPArtifact.INITIATOR_ROLE,
12         new Fact("accept"),
13         new CompositeExpression(LogicalOperatorType.OR,
14             new Fact("done"), new Fact("failure")))
15     }
16     );
17   }
```

```
18    // ...
19 }
```

Listing 10: Definition of `InitiatorRequirements`.

The class adds (lines 5–8, Listing 10) commitment *C(i, p, propose, accept* $\vee$ *reject)* (see Example 2) to the array `D_DEFINITION`, inherited from the superclass `RoleRequirements`. Meaning that any agent willing to play the *initiator* role must be debtor-compliant with the commitment above. At the same time, (lines 11–14) the class adds the commitment *C(p, i, accept, done* $\vee$ *failure)* to the array `C_DEFINITION`, again inherited from `RoleRequirements`, meaning that a role player will need to be creditor-compliant with such a commitment.

We complete this discussion with an example of behavior typing on the agent's side. An agent, that wills to play the *initiator* role, must offer a set of behaviors that are typed in a proper way. In Example 6 we have seen that these behaviors are *bvr-cfp*, *bvr-accept*, and *bvr-reject* (abstracted by `InitiatorBehaviour` in figures 3 and 4). Let us see how *bvr-accept* is actually defined and typed. First of all, the following listing shows how the implementation of *bvr-accept* is associated (via annotation) with its type:

```
1 @BehaviourType(bType = TypeBvrAcceptInitiator.class)
2 public class BvrAccept extends OneShotBehaviour
3     implements CNPInitiatorObserver {
4   // ...
5 }
```

Listing 11: Definition of class `BvrAccpet` implementing initiator's behavior *bvr-accept*.

The `TypeBvrAcceptInitiator` type is actually defined in the following listing where, as noted above, a b-type is implemented as a triple: commitment, partial actualization so far, and event that makes the commitment progress. The type in Listing12 follows such a structure by invoking the superclass constructor (i.e., `BType`): from line 3 to 9 the commitment $C(i, p, propose, accept \vee reject)$ is specified, in line 10 the current (empty) partial actualization is given, and finally, in line 11 the protocol action, invoked by the associated behavior, is specified.

```
1 public class TypeBvrAcceptInitiator extends BType {
2   public TypeBvrAcceptInitiator() {
3     super(new Commitment[]{
4       new Commitment(
5       new RoleId(CNP.INITIATOR_ROLE, RoleId.GROUP_ROLE),
6       new RoleId(CNP.PARTICIPANT_ROLE, RoleId.GROUP_ROLE),
7       new Fact("propose"),
8       new CompositeExpression(LogicalOperatorType.OR,
9       new Fact("accept"), new Fact("reject")))},
10      null,
11      new ProtocolAction("accept")
12    );
13  }
14  // ...
15 }
```

Listing 12: Definition of `TypeBvrAcceptInitiator`.

	Early Appr.	Global S. T.	FAAL	SimpAL	Our Appr.
protocol-based	✓	✓			✓
agent typing	✓		✓	✓	✓
system typing	✓	✓	✓	✓	✓
static type checking			✓	✓	
dynamic type checking	✓				✓
no disclosure		✓			✓
msg or meaning based	meaning	msg		msg	meaning
flexible					✓
agent concepts	✓				✓
struct. or behavioral	behavioral		structural	structural	behavioral

Table 4: Comparison of the relevant literature with our proposal.

6 Related Works

While a main research issue in concurrency programming, including the actor computing model [1], is the definition of formal models, calculi, and type systems analogously to the sequential case, for what concerns agents, just a few studies were carried on aspects such as typing and type safety. 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: supporting the development of error-free and human-readable code. On the other hand, most agent system implementations (JADE [18], Jack [41], A-Globe [63]) are based on programming languages like Java, and do not supply agent-oriented types, but rather rely on the typing system of the host language used for developing the system itself.

This section explains the main proposals concerning agent typing with their pros and cons. Table 4 provides a comparison at a glance. Each column corresponds to one of the introduced works, while each line is a feature that may or may not characterize the proposal. For helping understand where our proposal sits, the last column summarizes the key features of the typing we propose. Concerning the rows, a little explanation is needed. *Protocol-based* is checked for those proposals that concern protocol-ruled agent interaction. *Agent typing* is checked when the proposal provides agent typing. *System typing* says if the proposal provides typing of the system as a whole. In such case, system typing tendentiously concerns the interaction protocol. The *static type checking* and *dynamic type checking* rows indicate the kind of type checking that is supported. *No disclosure* is checked when the proposal does not require the disclosure of the involved agents' behavior. *Msg or meaning based* specifies if the typing is defined at the level of messages or at the level of message meaning. *Flexible* means that the typing does not capture a behavior procedurally. *Agent concepts* is marked when the typing bases upon concepts that are characteristic of the agent framework. Finally, *struct. or behavioral* indicates whether the typing system is concerned with the structure of the agents (or the system) or with its behavior.

Early approaches. To the best of our knowledge, Zapf and Geihs [71] 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. They 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 [48]. The third and last component is the *semantic type*, an annotation aimed at checking behavior-compatibility, based on J. F. Sowa's conceptual graphs.

This proposal shows many interesting features. It supports dynamic type checking, which means that the verification that an agent fits the requirements for interacting in a MAS is done only when the agent decides to enter the interaction. This is a pro because, in general, an agent may acquire the required properties just before it enters the system. Anticipating the check to earlier phases of the agent's life may result overly prescriptive. Another pro is that the typing relies only on externally visible actions, because the agents' internal states are not inspectable. Then, the proposal accounts for the interaction protocol, i.e. it captures the rules of encounter of the agents. On the negative side, the solution relies on finite state automata for describing both the interaction and the agents' behavior. This choice hinders the agent's autonomy in two ways. First, agents must supply a description of their behavior, a requirement that limits applicability. The reason is that in many practical settings, principals do not wish to disclose their internal procedures or, as in the case when legacy software is used, it may be impossible for them to disclose it. Second, 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 executing certain tasks in different ways. Prescribing precise sequences of actions limits flexibility. The agent will not be capable of coping with exceptional circumstances, errors/faults, opportunities, nor to adapt to dynamic conditions unless such alternative behaviors are explicitly listed among those accepted by the automaton. Quoting Cherns [25] "..., it is a mistake to specify more than is needed because by doing so options are closed that could be kept open. This premature closing of options is a pervasive fault in design."

Global session types for legacy software. The main claim of [5] is the importance of using interaction protocols for representing the functioning of a system. To this aim, the authors use global session types as an abstraction tool, which allows automatically generating monitor agents that are aimed at verifying the correctness of on-going, multi-party interactions. The monitor agent intercepts all the exchanged messages and verifies whether the protocol is respected. Like the previous proposal, also [5] focuses on externally visible actions (message exchanges) and on the use of interaction protocols. Moreover, similarly to finite state automata, global session types have a prescriptive, procedural nature, whose drawbacks have been explained above. The difference is that [5] provides no actual type system; rather global session types are used for specifying the interaction within a system from a global perspective. The proposal is implemented in Jason [23]. Global session types are represented by cyclic Prolog terms, which are consumed as messages are sniffed.

On the negative side, although relying on a monitor agent which checks the exchanged messages is an important functionality, that allows overlaying the verification of the interaction on top of legacy software (as it was the case in the cited work), it is hardly generalizable. Then, since agents are not typed and there is no expression of what an agent can or is expected to do, the proposal is more of a kind of run-time compliance checking. Another consequence is that when agents enter a system, it is not possible to verify whether their behavior is compatible with the protocol before the interaction begins. It is also impossible

to search for agents showing characteristics which allow them to successfully take part to the interaction.

Concerning the monitor, we disagree with the choice of realizing it 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. Centralization also becomes a bottleneck when the number of interacting agents grows. First steps for allowing the distribution of the verification of the on-going interactions are presented in [3], where an algorithm is described to project a global session type onto subsets of agents. Further developments are reported in [4] but they basically concern protocol switch as a way for agents to be adaptive.

Featherweight Agent and Artifact Language. The Featherweight Agent and Artifact Language (FAAL) [33] is a formal calculus that models the agent and artifact program abstractions provided by SimpA [56], a language that realizes all the relevant features of the A&A meta-model [68, 50]. A&A extends the agent paradigm with another primitive abstraction, the *artifact*: a computational, programmable system resource, that can be manipulated by agents, residing at the same abstraction level of the agent abstraction. A&A provides ways for defining workspaces, i.e., logical groups of artifacts, that can be joined by agents at runtime. Here agents can create, use, share and compose artifacts to support individual and collective, cooperative or antagonistic activities. The environment is itself programmable, and encapsulates services and functionalities.

FAAL accounts both for agents and for artifacts. A well-formed system configuration is seen as a parallel composition of *agent instances*, keeping track of a tree of activities to be carried on autonomously, and of *artifact instances*, holding a set of operations to be executed as responses to agent actions over the artifact instance. Agent and artifact instances are interpreted as independent and asynchronous processes. The calculus provides a basis on top of which the authors developed a type system that ensures the standard properties of progress and of preservation of well-formed configurations. It enjoys the type soundness property of statically typed languages. Well-formed programs do not get stuck in the sense that if an agent has some activity (or an artifact has some operation) to perform, then there is surely some rule that can be applied.

Although [33] is the first example of a formal account of agent (and artifact) typing, it is essentially close to a traditional data type system: each agent (artifact) is associated with a type that is based on its own structure, without taking into account the behavioral (state) evolution of the agent (artifact) itself. An agent is not a mere set of functions but rather it is determined by its capabilities, that can vary along with the execution also as a consequence of the relationships it entertains with other agents, for instance by playing different roles in relation with other agents. In addition, the calculus does not account for any of those high-level cognitive concepts that typically characterize agents and multi-agent systems, such as beliefs, desires, and intentions, relying instead on traditional functional abstractions.

Typing in SimpAL. In the SimpAL language [53–55] types are seen as useful for realizing integrated development environments, so much that the authors implemented an Eclipse plugin [58]. 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 behavioral

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, 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 [71], where the authors explain how conventional typing does not suffice the context of agent systems. The critic bases upon work by Nierstrasz [48] 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 a programming element, that is used by agents in order to take decisions.

7 Discussion and Future Work

This paper presented a typing system for MAS. The key characteristic of the proposal is that the typing system is based on notions that are typical of agents rather than on a functional approach. Specifically, it relies on the direct use of relationships among agents intended as first-class entities. As such, the proposal represents a novelty w.r.t. previous work on agent typing, which applies the functional type theory [37,38,54,33]. 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.

The proposal we have made takes a perspective on modeling multiagent systems that is akin to the one taken by the *artifact-centric approach* [20,29]. The artifact-centric approach counterposes a data-centric vision to the classical activity-centric vision in the specification of processes. An artifact is a concrete, identifiable, self-describing chunk of information, the basic building blocks by which business models and operations are described. In particular, it includes a lifecycle model, that contains the key states through which the data evolve, together with their transitions (triggered by the execution of corresponding tasks). In our vision, the social state of a protocol execution is the artifact. The lifecycle of such an artifact is given by the protocol. Theorem 1 together with Proposition 1 show that, in presence of a socially-progressive commitment protocol, it is possible to take into account just the commitment lifecycles when programming the agent behaviors.

To better clarify this point, let us consider the expression $z = f(g(x))$, i.e. $y = g(x); z = f(y)$. The classical verification carried on through a typing system verifies that the type of x is correct with respect to the type expected by g, that the value returned by g is correct with respect to the one expected by f, and finally that the value returned by f corresponds to the one of z. In an artifact-centric perspective, x is a datum that is transformed through the application of g followed by f. In the data-centric approach [49], there is no such input-output flow through calls of f and g but rather the information itself evolves. So, with reference to Figure 5, an artifact, initially amounting to the information that above was contained in x, would evolve to another state and then another, respectively amounting to the information

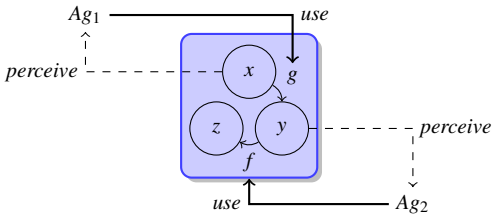

Fig. 5: A data-oriented representation of the composition of two functions, $y = g(x)$, $z = f(y)$.

that previously we put in y and z. The evolution would be caused by the execution of the operations f and g by the involved agents, as a consequence of the perception of certain artifact states. In Figure 5, for instance, Ag_1 perceives the artifact is in state x and uses the artifact, performing g. Ag_2 executes f when it perceives the state y of the artifact. In such a context, type verification becomes a verification of the coordination among the agents in using the artifact. This is the kind of verification our typing system allows. To this end, we use commitments as requirements. On the creditor side, they specify the opportunities an agent can take bringing about certain conditions, thanks to an obligation put on the debtor agent. On the debtor side, commitments specify the need of having certain behaviors that will allow them to accomplish the possible obligations they may be subject to.

This perspective of typing shares some aspects with the one studied (in object-oriented settings) by Typestate-Oriented Programming, introduced in [2] and developed in works like [32,31]. This is an extension of object-oriented programming where objects are modeled in terms of classes and of changing *states*. The authors propose that each state may have its own representation and methods which may cause an object transition to a new state. The current state of an object is tracked thanks to a flow-sensitive, permission-based type system. The introduction of states is motivated by the fact that it helps library writers and users to design, document, and reuse libraries more effectively, with new opportunities to expressiveness. Damiani *et al.* [32] propose to use the *state of an object* as a *coordination means* in concurrent object-oriented languages. Following Philippsen's terminology [51], the proposal realizes a form of *coordination on the side of the callee*, i.e., it implements coordination in the class that is accessed concurrently. The state class construct is introduced to design coordination protocols via objects that can be safely accessed in a concurrent way. All fields of such classes are private and their methods are synchronized (they execute in mutual exclusion on the receiver object). A state class specifies a collection of states as well as how methods make object states transition. Following again [51], the state class construct is a boundary coordination mechanism as it realizes callee-side coordination *by design*. An interesting point about callee-side coordination is that it is possible to reason about the correctness of a class implementation based on local information, since all coordination code is part of the class implementation. In our proposal, the lifecycle is represented by the commitment protocol itself in a declarative way thanks to commitments, which provide also the means for specifying role requirements. Moreover, the identification of the class of socially-progressive protocols allows standardizing state transitions, that agents should tackle, in terms of commitment state changes.

Relationships and roles are key elements of the proposal. Since the early '80s, indeed, Rumbaugh [57] identified the notion of relationship as a *first-class* element, complementary to that of object, that cannot be obtained by relying on other notions. Steinmann [65] then proposed, for Object-Oriented languages, that roles should be seen as *intermediaries*

between relationships and the objects that engage them. These concepts were further developed for multi-agent systems. In particular, by extending Sowa's ideas [64], Guarino [39] underlines that roles differ from so called "natural" types (i.e., those that relate to the essence) in that individuals can enter or leave roles without losing their identity, and that playing a role requires individuals to be in relationship with other individuals. In this perspective, the types that constitute the type system proposed in [33] for A&A resemble more the notion of natural type while our proposal stands at the level of roles and relationships. Still concerning A&A, notice also that artifacts are not institutions, although they may resemble them. They differ in how agents interact with them. While an artifact is a sort of shared object that is accessible to all agents that are in the environment, to which the artifact belongs, an institution is accessible only to those agents which have explicitly joined that institution by playing (at least) a role defined within that institution.

Dastani et al. [34] explore the two concepts of *agent role* and *agent type* that are at the core of methodologies for developing MASs; according to them, during the analysis phase organizational structures are defined in terms of agent roles, and at design time sets of agent roles are translated into agent types. Dastani et al. propose an approach to account for role dynamics by introducing the role operations enact, deact, activate, and deactivate. An *agent role* is considered as a triple including a set of normative behavioral rules (conditional and context-dependent), a set of expected objectives, and a specification of the information that is expected to be available to the role players. In different words, *obligations* (i.e., states that must be achieved by the role player), *goals* (i.e., states that the role player wants to achieve), and *information* (received by the player when enacting the role). The properties of role coherence, and of consistency of a set of roles are defined. An *agent type* is a consistent subset of a given set of agent roles. Roles are intended as guidelines for the specification, design and implementation of MASs. Whether or not an agent is playing a role is a social fact, and an agent cannot deact a role at any time (e.g., an agent playing the role buyer cannot deact the role before paying a bought item). An agent that *enacts* a role adopts the goals and rules of the role. Enacting a role, thus, modifies the belief base of an agent and indirectly affects its behavior. Nevertheless, to achieve the role's objectives the role should be *activated*. The agent is free to select the order according to which pursuing its own goals and those acquired via the enacted roles. On the other hand, deacting/deactivating a role means, respectively, that an agent stops enacting the role, and that an agent becomes passive with respect to the role's objectives. In our view, however, roles are not abstractions to be used only at design time, and players are not isolated agents: they interact with one another. So, we do not see agents as characterized by sets of roles but by their capabilities, and focus on the issue of understanding at enactment time whether they can fulfill the engagements they may take towards other agents along the interaction ruled by some reference protocol. To achieve this purpose, we do not represent roles in terms of goals, but rather in terms of the social relationships that can arise and tie the agents as they play the protocol roles. Notice, however, that our approach is not limited to perform a dynamic type checking but could be used, as in [34], also at design time along the lines of [8].

Besides providing the basic notions of agent type, role requirements, and check of a role enactment, we implemented the proposal in the context of the 2COMM framework [6]. 2COMM enables programming social relationships by exploiting a declarative, interaction-centric approach. 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 [48,71] that the typing system should include a representation of the behavior but, differently from 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 this paper we have adopted the precedence logic language for expressing antecedent and consequent conditions. As already explained, this choice is mainly motivated by our belief that, to the purpose of typing, the specification language should be kept as simple as possible in order not to burden a programmer with a too hard task when it comes the time of defining and implementing the types of agents' behaviors. In case a more expressive language would be required, e.g., for expressing metric time constraints, our proposal could be extended in various way. A possibility is to adopt an extension of commitments with deadlines, such as those proposed in [28,27,26]. An alternative approach would be to adopt approaches like 2CL [13], 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 (e.g. [19]). In particular, agents could operate in more complex normative scenarios where, besides commitments, also authorizations and prohibitions could be present. For instance, in Revani [42], a MAS is specified in terms of a set of norms (i.e., commitments, authorizations, and prohibitions), which are added, or updated, incrementally in order to meet some predefined user requirements. The user requirements are encoded as CTL formulae whose validity is checked against the current set of norms. Whenever a requirement is not satisfied, the designer adjusts the current set of norms either by adding a new norm, or by revising some of those already defined – norm specification patterns are proposed as an aid for the designer. The process terminates when all the requirements are satisfied by the norms at hand; these norms are a specification of the sought MAS. Under this respect, Revani is complementary to our enactment checking because it focuses on the correctness of the system where the agents will operate. The checking we proposed concerns instead the adequacy of an agent in playing a role within a system.

Type checking, as a form of lightweight verification, adopts notions (e.g., substitutability), that are used also for coping with issues of interoperability and conformance discussed in [14,11]. 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. Another direction of research that we intend to explore is how commitment-based types can be adapted to solve the issue of conformance in MAS.

The described agent typing system will help realizing both static, compile-time coding support, and dynamic, run-time type checking. Inspired by [58], the former could be realized by developing a plug-in for an IDE that provides coding support, like smart code completion or warnings about possible type mismatches. 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, and hence signaling the occurrence of wrong enactments when needed. Commitment representations that account for an explicit representation of time, including deadlines, as for instance [28], are also a possible future direction of development. Particularly interesting, in this respect, is the proposal in [43], which aims at detecting possible misuses of a MAS, again specified as a set of norms. In this case, norms are expressed

in event calculus, which allows a designer to associate norms with a metric time setting the horizon within which they have to be satisfied. A simple Prolog program can thus infer all possible misuses of the system looking at all possible runs that either violate some norms, or are not conformant with domain-dependent rules. Knowing the possible misuses, a designer can decide to refine some of the norms to prevent, if possible, the misuses, or place some monitors to detect these misuses when the system will be up and running.

Acknowledgements This work was partially supported by the *Accountable Trustworthy Organizations and Systems (AThOS)* project, funded by Università degli Studi di Torino and Compagnia di San Paolo (CSP 2014). We would like to thank the reviwers for their comments and the discussions which holped to improve the work.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In S. Arora and G. T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 1015–1022. ACM, 2009.
3. 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 M. B. van Riemsdijk, editors, *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, volume 8758 of *Lecture Notes in Computer Science*, pages 246–270. Springer, 2014.
4. D. Ancona, D. Briola, A. Ferrando, and V. Mascardi. Global Protocols as First Class Entities for Self-Adaptive Agents. In G. Weiss, P. Yolum, R. H. Bordini, and E. Elkind, editors, *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2015, Istanbul, Turkey, May 4-8, 2015*, pages 1019–1029. ACM, 2015.
5. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic generation of self-monitoring mass from multiparty global session types in jason. In M. Baldoni, L. Dennis, V. Mascardi, and W. Vasconcelos, editors, *Declarative Agent Languages and Technologies X*, volume 7784 of *Lecture Notes in Computer Science*, pages 76–95. Springer Berlin Heidelberg, 2013.
6. M. Baldoni, C. Baroglio, and F. Capuzzimati. A Commitment-Based Infrastructure for Programming Socio-Technical Systems. *ACM Transactions on Internet Technology*, 14(4):23:1–23:23, Dec. 2014.
7. M. Baldoni, C. Baroglio, and F. Capuzzimati. Typing Multi-Agent Systems via Commitments. In F. Dalpiaz, J. Dix, and M. B. van Riemsdijk, editors, *Post-Proc. of the 2nd International Workshop on Engineering Multi-Agent Systems, EMAS 2014, Revised Selected and Invited Papers*, number 8758 in LNAI, pages 388–405. Springer, 2014.
8. M. Baldoni, C. Baroglio, F. Capuzzimati, and R. Micalizio. Empowering Agent Coordination with Social Engagement. In M. Gavanelli, E. Lamma, and F. Riguzzi, editors, *AI*IA 2015: Advances in Artificial Intelligence, XIV International Conference of the Italian Association for Artificial Intelligence*, volume 9336 of *LNAI*, pages 89–101, Ferrara, Italy, September 2015. Springer.
9. M. Baldoni, C. Baroglio, F. Capuzzimati, and R. Micalizio. Exploiting Social Commitments in Programming Agent Interaction. In Q. Chen, P. Torroni, S. Villata, J. Y. Hsu, and A. Omicini, editors, *PRIMA 2015: Principles and Practice of Multi-Agent Systems, 18th International Conference*, number 9387 in Lecture Notes in Computer Science, pages 566–574, Bertinoro, Italy, October 26th–30th 2015. Springer.
10. M. Baldoni, C. Baroglio, F. Capuzzimati, and R. Micalizio. Commitment-based Agent Interaction in JaCaMo+. *Fundamenta Informaticae*, 157:1–33, 2018.
11. M. Baldoni, C. Baroglio, A. K. Chopra, N. Desai, V. Patti, and M. 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.
12. M. Baldoni, C. Baroglio, A. K. Chopra, and M. P. Singh. Composing and Verifying Commitment-Based Multiagent Protocols. In M. Wooldridge and Q. Yang, editors, *Proc. of 24th International Joint Conference on Artificial Intelligence, IJCAI 2015*, Buenos Aires, Argentina, July 25th-31th 2015.
13. M. Baldoni, C. Baroglio, E. Marengo, and V. 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.

14. M. Baldoni, C. Baroglio, A. Martelli, and V. 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.
15. M. Baldoni, G. Boella, and L. van der Torre. Interaction between Objects in `powerjava`. *Journal of Object Technology, Special Issue OOPS Track at SAC 2006*, 6(2), 2007.
16. M. Baldoni, G. Boella, and L. van der Torre. Relationships Meet their Roles in Object Oriented Programming. In F. Arbab, A. Movaghar, J. Rutten, and M. Sirjani, editors, *Proc. of the International Symposium on Fundamentals of Software Engineering, FSEN'07*, volume 4767 of *Lecture Notes in Computer Science (LNCS)*, pages 440–448, Tehran, Iran, April 2007. Springer.
17. M. Baldoni, G. Boella, and L. van der Torre. The Interplay between Relationships, Roles and Objects. In F. Arbab, H. Sarbazi-azad, and M. Sirjani, editors, *Proc. of the International Conference on Fundamentals of Software Engineering, FSEN'09*, volume 5961 of *Lecture Notes in Computer Science (LNCS)*, pages 402–415, Kish Island, Persian Gulf, Iran, April 2009. Springer.
18. 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.
19. J. Bentahar, J.-J. C. Meyer, and W. Wan. Model Checking Communicative Agent-based Systems. *Knowledge-Based Systems*, 22(3):142–159, 2009.
20. K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.
21. G. Boella and L. W. N. van der Torre. The Ontological Properties of Social Roles in Multi-Agent Systems: Definitional Dependence, Powers and Roles playing Roles. *Artificial Intelligence and Law Journal (AILaw)*, 15(3):201–221, 2007.
22. O. Boissier, R. H. Bordini, J. F. Hübner, A. Ricci, and A. Santi. Multi-Agent Oriented Programming with JaCaMo. *Science of Computer Programming*, 78(6):747 – 761, 2013.
23. R. H. Bordini and J. F. Hübner. BDI Agent Programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer Berlin Heidelberg, 2006.
24. 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.
25. A. Cherns. Principles of Socio-Technical Design. *Human Relations*, 2:783–792, 1976.
26. F. Chesani, P. Mello, M. Montali, and P. Torroni. Monitoring time-aware social commitments with reactive event calculus. In *Proceedings of the 7th international symposium From Agent Theory to Agent Implementation (AT2AI-7)*, 2010.
27. F. Chesani, P. Mello, M. Montali, and P. Torroni. Monitoring time-aware commitments within agent-based simulation environments. *Cybernetics and Systems*, 42(2):546–566, 2013.
28. F. Chesani, P. Mello, M. Montali, and P. Torroni. Representing and Monitoring Social Commitments Using the Event Calculus. *Autonomous Agents and Multi-Agent Systems*, 27(1):85–130, 2013.
29. D. Cohn and H. Richard. Business Artifacts: A Data-centric Approach to Modeling Business Operations and Processes. *IEEE Data Eng. Bull.*, 32(3):3–9, 2009.
30. M. Colombetti, N. Fornara, and M. Verdicchio. The role of institutions in multiagent systems. In *In Proc. of the Workshop on Knowledge-Based and Reasoning Agents, VIII Convegno AI*IA*, pages 118–2, 2002.
31. S. Crafa and L. Padovani. The chemical approach to typestate-oriented programming. In J. Aldrich and P. Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 917–934. ACM, 2015.
32. F. Damiani, E. Giachino, P. Giannini, and S. Drossopoulou. A type safe state abstraction for coordination in java -like languages. *Acta Inf.*, 45(7-8):479–536, 2008.
33. F. Damiani, P. Giannini, A. Ricci, and M. Viroli. Standard Type Soundness for Agents and Artifacts. *Scientific Annals of Computer Science*, 22(2):267–326, 2012.
34. M. Dastani, M. B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. C. Meyer. Enacting and deacting roles in agent programming. In J. Odell, P. Giorgini, and J. P. Müller, editors, *Agent-Oriented Software Engineering V*, volume 3382 of *Lecture Notes in Computer Science*, pages 189–204. Springer Berlin Heidelberg, 2005.
35. Y. Demazeau. From interactions to collective behaviour in agent-based systems. In *Proc. of the 1st. European Conference on Cognitive Science*, Saint-Malo, 1995.
36. G. Governatori. Law, logic and business processes. In *Third International Workshop on Requirements Engineering and Law, RELAW 2010, Sydney, NSW, Australia, September 28, 2010*, pages 1–10. IEEE, 2010.

37. C. Grigore and R. Collier. Supporting Agent Systems in the Programming Language. In J. F. Hübner, J.-M. Petit, and E. Suzuki, editors, *Web Intelligence/IAT Workshops*, pages 9–12. IEEE Computer Society, 2011.
38. C. Grigore and R. W. Collier. AF-Raf: an Agent-Oriented Programming Language with Algebraic Data Types. In *SPLASH Workshops*, pages 195–200, 2011.
39. N. Guarino. Concepts, Attributes, and Arbitrary Relations – Some Linguistic and Ontological Criteria for Structuring Knowledge Bases. *Data & Knowledge Engineering*, 8:249–261, 1992.
40. N. Guarino and C. A. Welty. Evaluating Ontological Decisions with OntoClean. *Communications of the ACM*, 45(2):61–65, 2002.
41. N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Summary of an Agent Infrastructure. In *Proc. of the 5th International Conference on Autonomous Agents*, 2001.
42. Ö. Kafali, N. Ajmeri, and M. P. Singh. Revani: Revising and verifying normative specifications for privacy. *IEEE Intelligent Systems*, 31(5):8–15, 2016.
43. Ö. Kafali, M. P. Singh, and L. A. Williams. NANE: identifying misuse cases using temporal norm enactments. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*, pages 136–145, 2016.
44. N. Kökciyan and P. Yolum. Priguard: A semantic approach to detect privacy violations in online social networks. *IEEE Trans. Knowl. Data Eng.*, 28(10):2724–2737, 2016.
45. N. Kökciyan and P. Yolum. Priguardtool: A tool for monitoring privacy violations in online social networks (demonstration). In C. M. Jonker, S. Marsella, J. Thangarajah, and K. Tuyls, editors, *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9-13, 2016*, pages 1496–1497. ACM, 2016.
46. E. Marengo, M. Baldoni, C. Baroglio, A. K. Chopra, V. Patti, and M. 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.
47. C. Masolo, L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi, and N. Guarino. Social Roles and their Descriptions. In D. Dubois, C. A. Welty, and M. Williams, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*, pages 267–277. AAAI Press, 2004.
48. O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition*, chapter 6, pages 99–121. Prentice Hall, 1995.
49. A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3):428 – 445, 2003.
50. A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A Meta-Model for Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, 2008.
51. M. Philippsen. A survey of concurrent object-oriented languages. *Concurrency - Practice and Experience*, 12(10):917–980, 2000.
52. A. Ricci, M. Piunti, and M. Viroli. Environment Programming in Multi-Agent Systems: an Artifact-based Perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.
53. A. Ricci and A. Santi. From Actors to Agent-Oriented Programming Abstractions in simpAL. In *SPLASH Workshops*, pages 73–74, 2012.
54. A. Ricci and A. Santi. Typing Multi-agent Programs in simpAL. In M. Dastani, J. F. Hübner, and B. Logan, editors, *Programming Multi-Agent System*, volume 7837 of *Lecture Notes in Computer Science*, pages 138–157. Springer, 2012.
55. A. Ricci and A. Santi. From Actors and Concurrent Objects to Agent-Oriented Programming in simpAL. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond - Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday*, volume 8665 of *Lecture Notes in Computer Science*, pages 408–445. Springer, 2014.
56. A. Ricci, M. Viroli, and G. Piancastelli. simpA: An Agent-Oriented Approach for Programming Concurrent Applications on top of Java. *Science of Computer Programming*, 76(1):37–62, 2011.
57. J. Rumbaugh. Relations As Semantic Constructs in an Object-oriented Language. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 466–481, New York, NY, USA, 1987. ACM.
58. A. Santi and A. Ricci. An Eclipse-based IDE for Agent-Oriented Programming in simpAL. In *Proc. of The Seventh Workshop of the Italian Eclipse Community*, 2012.
59. M. P. Singh. An Ontology for Commitments in Multiagent Systems. *Artificial Intelligence and Law Journal (AILaw)*, 7(1):97–113, 1999.
60. M. P. Singh. A Social Semantics for Agent Communication Languages. In F. Dignum and M. Greaves, editors, *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2000.

61. M. P. Singh. Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition. In *The Second International Joint Conference on Autonomous Agents & Multiagent Systems, AAMAS 2003, July 14-18, 2003, Melbourne, Victoria, Australia, Proceedings*, pages 907–914. ACM, 2003.
62. M. P. Singh and M. N. Huhns. *Service-oriented computing - semantics, processes, agents*. Wiley, 2005.
63. D. Šišlák, M. Rehák, M. Pěchouček, M. Rollo, and D. 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.
64. J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
65. F. Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83 – 106, 2000.
66. SUN Microsystems, Inc. JSR 175: A Metadata Facility for the Java Programming Language, 2002. https://jcp.org/en/jsr/detail?id=175.
67. P. R. Telang, M. P. Singh, and N. Yorke-Smith. Relating goal and commitment semantics. In *ProMAS*, volume 7217 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2011.
68. D. Weyns, A. Omicini, and J. Odell. Environment as a First Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.
69. P. Yolum and M. P. Singh. Designing and executing protocols using the event calculus. In *Proceedings of the Fifth International Conference on Autonomous Agents*, AGENTS '01, pages 27–28, New York, NY, USA, 2001. ACM.
70. P. Yolum and M. P. Singh. Commitment Machines. In *Intelligent Agents VIII, 8th International Workshop, ATAL 2001*, volume 2333 of *LNCS*, pages 235–247. Springer, 2002.
71. M. Zapf and K. Geihs. What Type Is It? A Type System For Mobile Agents. In *15th European Meeting on Cybernetics and Systems Research (EMCSR)*, 2000.