Empowering Agent Coordination with Social Engagement

Matteo Baldoni(✉), Cristina Baroglio, Federico Capuzzimati, Roberto Micalizio

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

Abstract. Agent coordination based on Activity Theory postulates that agents control their own behavior from the outside by using and creating artifacts through which they interact. Based on this conception, we envisage social engagements as first-class resources that agents exploit in their deliberative cycle (as well as beliefs, goals, intentions), and propose to realize them as artifacts that agents create and manipulate along the interaction, and that drive the interaction itself. Consequently, agents will base their reasoning on their social engagement, instead of relying on event occurrence alone. Placing social engagement at the center of coordination promotes agent decoupling and also the decoupling of the agent specifications from the specification of their coordination. The paper also discusses JaCaMo+, a framework that implements this proposal.

Keywords: Social Engagement, Commitments, Agents and Artifacts, Agent Programming, Implementation.

1 Introduction

We propose an agent programming approach that is inspired by, and extends, the *environment programming* methodology proposed in [18]. In this methodology, the environment is seen as a programmable part of the (multiagent) system. The methodology takes advantage of the A&A meta-model [23,17] that, having its roots in the Activity Theory [13], extends the agent paradigm with the *artifact* primitive abstraction. An artifact is a computational, programmable system resource, that can be manipulated by agents.

In this context, programming a Multiagent System (MAS) consists of two main activities: (1) programming the agents as autonomous software entities designed to accomplish user-defined goals; and (2) programming the environment (i.e., the artifacts) that provide agents with those functionalities they can exploit while performing their own activities. In other words, the environment where the agents operate is thought of as a set of artifacts, where each artifact is a resource, or tool, that is made available to the agents. Indeed, the framework is much more powerful as it allows agents not only to access and use artifacts, but also to create new artifacts, to adapt them, and even to link two (or more) artifacts.

An artifact provides the agents using it with a set of *operations* and a set of *observable properties*. Operations are computational processes that are executed inside the artifact itself, and that can be triggered by agents or other artifacts. Observable properties are state variables that are observable by all those agents using, or *focusing* on, the artifact. Of course, observable properties can change over time as a result of the operations occurred inside the artifact. It must be noticed that, although artifacts are substantially a coordination mechanism, direct communication between agents is still possible in the framework proposed by Ricci et al. [18]

The *Coordination by Social Engagement* (CoSE) programming approach proposed in this paper is a characterization of the A&A meta-model that aims at further simplifying the design and programming of MASs. First of all, in CoSE agents are not allowed to communicate directly, not even via message exchange; agent interaction can occur only by way of artifacts (as postulated by Activity Theory [13]). In addition, CoSE better characterizes the content of artifact observable properties. These properties are not only (shared) state variables, but are also, and more importantly, *social relationships*. Namely, structures that explicitly represent the dependencies existing between any two agents that interact through a same artifact.

In this paper we show how the programming of interacting agents can be systematically approached by relying on the explicit representation of the agents' *social engagement*. The basic idea is that when agents can directly handle social relationships as *resources*, the coding phase can be organized in a precise sequence of steps. Specifically, we focus on social relationships that can be captured as *social commitments* [19]. The advantages are both on the *software engineering* perspective (decoupling of code), and on the *modeling* perspective (agents may consider truly social dependencies, and thus other agents, in their deliberative cycle).

2 Coordination via Social Engagement

We consider social relationships that can be represented as *social commitments* [19]. A commitment $\mathsf{C}(x, y, s, u)$ captures that agent x (debtor) commits to agent y (creditor) to bring about the consequent condition u when the antecedent condition s holds. Antecedent and consequent conditions are conjunctions or disjunctions of events and commitments. Commitments have a *life cycle*. We adopt the commitments life cycle proposed in [21]. Briefly, a commitment is *Null* right before being created; *Active* when it is created. Active has two substates: *Conditional* (as long as the antecedent condition did not occur), and *Detached* (when the antecedent condition occurred). In the latter case, the debtor is now engaged in the consequent condition of the commitment. An Active commitment can become: *Pending* if suspended; *Satisfied*, if the engagement is accomplished; *Expired*, if it will not be necessary to accomplish the consequent condition; *Terminated* if the commitment is canceled when Conditional or released when Active; and finally, *Violated* when its antecedent has been satisfied, but its con-

sequent will be forever false, or it is canceled when Detached (the debtor will be considered liable for the violation). As usual, commitments are manipulated by the commitment operations *create* (an agent creates a commitment toward someone), *cancel* (a debtor withdraws an own commitment), *release* (an agent withdraws a commitment of which it is the creditor), *assign* (a new creditor is specified by the previous one), *delegate* (a new debtor is specified by the previous one), *discharge* (consequent condition u holds). Since debtors are expected to behave so as to satisfy their engagements, commitments create social expectations on the agents' behaviors [9]. Moreover, since we implement commitments as observable properties of an artifact, they can be used by agents in their practical reasoning together with beliefs, intentions, and goals for taking into account other agents and the conditions the latter committed to have achieved.

Programming a MAS in our setting requires, as in the environment programming methodology, to program both *agents* and *artifacts*. CoSE provides a *commitment-driven methodology* for programming agents.

Artifacts. We consider artifacts that include commitments among their observable properties; whenever this happens, we say that the artifact has a *social state*. Thus, an artifact Art is formally represented as a tuple $\langle S, O, R, \rho \rangle$, where:

1. S is a social state, namely, a set of state variables and social relationships represented as commitments;
2. O is a set of artifact operations made available to the agents focusing on the artifact; each operation op in O has a social meaning: The execution of op creates a new social relationship (i.e., creates a new commitment), or it makes an existing social relationship evolve (i.e., the state of a commitment changes). In other words, whenever an artifact operation is executed, the artifact social state changes not only because some state variables change their values, but also because some social relationships evolve (e.g., new commitments are created, or existing commitments are detached/satisfied). Along the line discussed in [2], we say that the type τ of an operation op, $op : \tau$, is given by the set $\{c_1, \ldots, c_n\}$ of commitments that are created by the execution of op.
3. R is a set of role names exposed by the artifact: an agent which intends to use the artifact must enact one the roles it exposes.
4. $\rho : R \rightarrow 2^O$ is a function mapping role names to subset of operations in O; for each $r \in R$, $\rho(r)$ denotes the subset of operations that an agent playing role r can legally perform on the artifact.

Since operations are typed, and since roles map to operations, we can associate each role $r \in Art.R$ with a type. Formally, a role $r \in Art.R$ has a type τ, $r : \tau$, such that

$$\tau = \bigcup_{op_i \in \rho(r) | op_i : \tau_i} \tau_i,$$

namely, the type of a role is the union of all the types of the operations that are associated with that role.

Agents. For the purposes of this paper, an agent Ag can be abstracted as a triple $\langle \Sigma, B, G \rangle$; where Σ is the agent internal state, inspectable by Ag only, and B is a set $\{b_1, \ldots, b_m\}$ of *behaviors*, each of which enables Ag to perform a given activity. In other terms, each behavior represents a piece of software that the designer foresees in the agent specification phase, and, then, the programmer actually implements in the development phase. Also behaviors can be associated with types. According to [2], a behavior b has type τ, denoted as $b : \tau$, where τ is, as before, a set of commitments $\{c_1, c_2, \ldots, c_n\}$. A behavior b of type τ is capable of satisfying the commitments in the type. [2] also pointed out how the usage of typed behaviors enables a dynamic type-checking of agents, guaranteeing that an agent can only enact roles for which it can satisfy all the involved commitments. Finally, G is a set of goals assigned to the agent.

CoSE Methodology. Programming an agent Ag, thus, comes down to implementing the set of its behaviors. To this aim, the CoSE methodology suggests a programmer the following steps. Let $\mathcal{E}nv$ be a programming environment consisting of a set $\{Art_1, \ldots, Art_n\}$ of artifacts, let G be a set $\{g_1, \ldots, g_k\}$ of goals assigned to Ag, and let B be the initial (possibly empty) set of behaviors of Ag. For each goal $g_i \in G$:

1. If g_i can be obtained by Ag without the need of interacting with other agents, then, program the behavior b for achieving g_i and add b to B;
2. Otherwise, Ag needs to interact with other agents. To this aim, select a suitable artifact Art_i in $\mathcal{E}nv$. This choice is made by relying on the artifact roles and operations. Specifically, the following matches are considered:
   (a) $\exists op \in Art_i.O$ *whose social meaning is create*$(\mathsf{C}(x, y, g_i, p))$. Intuitively, an agent playing the role x *offers* to have the consequent condition p achieved if some other agent, playing role y, will have g_i achieved. Thus, if Ag will play the role x, by performing op it will become the debtor of a conditional commitment, created in the social state $Art_i.S$. If another agent, playing role y, will bring about the antecedent g_i (i.e., the goal Ag is interested in), Ag will be engaged in bringing about the consequent.
   (b) $\mathsf{C}(Ag', y, q, g_i) \in Art_i.S$ *and* $\exists op \in Art_i.O$*, with social meaning* detach$(\mathsf{C}(Ag', y, q, g_i))$. Intuitively, the social state $Art_i.S$ already contains an offer by agent Ag', and if Ag plays role y, then, by performing op, it will accept that offer. Thus, by achieving the antecedent condition q, Ag will bind the other agent to bring about g_i.
3. Once a suitable artifact Art_i has been selected, the programmer has to identify the role(s) in $Art_i.R$ agent Ag could play during its execution. Such a step is partially based on the set of operations that have been previously recognized as useful for achieving g_i. Note, however, that function ρ just maps roles to subsets of operations, but it does not induce a partition on $Art_i.O$; that is, given any two roles $r1$ and $r2$ in R, $\rho(r1) \cap \rho(r2)$ is not necessarily empty. Therefore, once the programmer has identified the operations Ag needs for achieving g_i, the programmer has to select, among all roles in $Art_i.R$ enabling such operations, a role r that better than others fits her/his needs. Let *Roles* be such a set of selected role(s) in Art_i.

4. For each role $r \in Roles$, let r be of type $\tau=\{c_1, \ldots, c_n\}$. Then, for each commitment $c_j \in \tau$, program a behavior b agent Ag assumes whenever a state change occurs on c_j, and add b in B.

Following these steps, the programmer will implement an agent incrementally, by considering one goal at a time, and by focusing on subproblems (i.e., behaviors), that either directly or via interaction will be programmed to obtain the goal at hand.

CoSE enjoys the following properties:

- *Agent-to-Agent Decoupling*: Since agents can only interact through artifacts, the separation between agents is even more neat than in the direct message exchange case. The exchange of messages, in fact, assumes that the two agents (the sender and the receiver) share a common language. In CoSE it is only required that agents are capable of using artifact operations. The advantage is to promote agent openness and heterogeneity.
- *Agent-Logic-to-Coordination-Logic Decoupling*: Programming an agent just requires the designer to consider two main aspects: (1) the domain-dependent process for reaching the goal the agent is devised for, and (2) the behavior of the agent as a response to changes in the social states of the artifacts the agent is focusing on. The coordination logic is no longer part of the agent; rather, the coordination logic is only implemented within the artifact. Among the main advantages, *code verification*, i.e. the interaction logic is programmed just in one precise portion of the system and verified once only, and *Code Maintainability*, i.e. the interaction logic is not spread across the agents, changes to the interaction logic just involve the artifact, while agents do not need to change.

3 JaCaMo+: Programming Coordination with Social Relationships

JaCaMo+ builds on the seminal work [1] and on *JaCaMo* [6], a platform integrating Jason (as an agent programming language), CArtAgO (as a realization of the A&A meta-model), and Moise (as a support to the realization of organizations). A MAS realized in JaCaMo is a Moise agent organization, which involves a set of Jason agents, all working in CArtAgO environments. CArtAgO environments can be designed and programmed as a dynamic set of shared artifacts, possibly distributed among various nodes of a network, that are collected into workspaces. By *focusing* on an artifact, an agent registers to be notified of events that are generated inside the artifact; e.g., when other agents execute some action.

Jason [7] implements in Java, and extends, the agent programming language AgentSpeak(L). Jason agents are characterized by a BDI architecture: Each of them has (1) its belief base, which is a set of ground (first-order) atomic formulas; and (2) its set of plans (plan library). It is possible to specify two types of goals:

achievement goals (atomic formulas prefixed by the '!' operator) and *test goals* (prefixed by '?'). Agents can reason on their beliefs/goals and react to events, amounting either to belief changes (occurred by sensing their environment) or to goal changes. Each plan has a triggering event (an event that causes its activation), which can either be the addition or the deletion of some belief or goal. The syntax is inherently declarative. In JaCaMo, the beliefs of Jason agents can also change due to operations performed by some agent of the MAS on the CArtAgO environment, whose consequences are automatically propagated.

JaCaMo+ extends JaCaMo along different directions. In particular, JaCaMo+ reifies the social relationships (commitments) as resources that are available to the interacting agents. This was obtained by enriching CArtAgO artifacts with an explicit representation of commitments and of commitment-based interaction protocols. In this way, JaCaMo+ seamlessly integrates Jason BDI agents with social commitments. The resulting class of artifacts reifies the execution of commitment-based protocols, including the social state, and enables Jason agents to be notified about the social events, and to perform practical reasoning about *social expectations* thanks to commitments: Agents expect that the commitment debtors behave so as to satisfy the corresponding consequent conditions, and use such information to decide about their own behavior and goals.

A *protocol artifact* is a JaCaMo+ artifact that implements a commitment-based protocol, structured into a set of roles, which can manipulate the protocol social state. By enacting a role, an agent receives "social powers", whose execution has public social consequences, expressed in terms of commitments. A JaCaMo+ agent, focusing on a protocol artifact, has access to the social state of the artifact. The implementation, actually, maps the social state onto a portion of the belief base each such agent has: any change occurred in the artifact's social state is instantaneously propagated to the belief bases of all the focusing agents. Agents are, thereby, constantly aligned with the social state.

An agent playing a role can only execute the protocol actions that are associated with such a, otherwise the artifact raises an exception that is notified to the violator. When a protocol action is executed, the social state is updated accordingly by adding new commitments, or by modifying the state of existing commitments. The artifact is responsible for maintaining the social state up-to-date, following action execution and the commitment life cycle.

JaCaMo+ allows specifying Jason plans, whose triggering events involve social relationships; i.e., commitments. In JaCaMo+, a commitment is represented as a term $cc(debtor, creditor, antecedent, consequent, status)$ where *debtor* and *creditor* identify the involved agents, while *antecedent* and *consequent* are the commitment conditions. As a difference with standard commitment notation, we explicitly represent a commitment state by means of the *Status* parameter. Commitments can be used inside a *plan context* or *body*. Differently than beliefs, commitment assertion/deletion can only occur through the artifact, as a consequence of a change of the social state. For example, this is the case that deals with commitment addition:

$$+cc(debtor, creditor, antecedent, consequent, status) : \langle context \rangle \leftarrow \langle body \rangle.$$

The plan is triggered when a commitment that unifies with the one specified in the plan head appears in the social state. The syntax is the standard for Jason plans. *Debtor* and *creditor* are to be substituted by the proper role names. The plan may be aimed at achieving a change of the commitment status (e.g., the debtor will try to bring about the consequent and satisfy the commitment), or at allowing the agent to do something as a reaction (e.g., collecting information). Similar schemata can be defined to tackle commitment deletion and the addition (deletion) of social facts. JaCaMo+ allows using commitments also in contexts and plans as test goals $?cc(\dots)$, or achievement goals $!cc(\dots)$. Addition or deletion of such goals can, as well, be managed by plans. For example:

$$+!cc(debtor, creditor, antecedent, consequent, status) : \langle context \rangle \leftarrow \langle body \rangle.$$

The plan is triggered when the agent creates an achievement goal concerning a commitment. Consequently, the agent will act upon the artifact so as to create the desired social relationship. After the execution of the plan, the commitment $cc(debtor, creditor, antecedent, consequent, status)$ will hold in the social state, and will be projected onto the belief bases of each agent focusing on the artifact.

4 CoSE Methodology in Action

We explain the impact of our proposals by comparing the JaCaMo implementation [18] of Dijkstra's Dining Philosophers, with an implementation obtained via the CoSE methodology in JaCaMo+. The problem involves two roles and, thus, two kinds of agents: *waiter*, which is in charge of initializing the artifact, and *philosopher*. (We omit *waiter* because trivial and not interactive.)

In the JaCaMo implementation, coordination is obtained by relying on a CArtAgO artifact. Goal start initializes information about the philosopher's forks, and starts the main loop (*thinking* and then *eating*).

```
1  !start.
2  +!start <- .my_name(Me);
3          in("philo_init",Me,Left,Right);
4          +my_left_fork(Left); +my_right_fork(Right);
5          !living.
6  +!living <- !thinking;
7           !eating;
8           !!living.
9  +!eating <- !acquireRes; !eat; !releaseRes.
10 +!acquireRes: my_left_fork(Left) & my_right_fork(Right)
11     <-  in("ticket");
12         in("fork",Left); in("fork",Right).
13 +!releaseRes: my_left_fork(Left) & my_right_fork(Right)
14     <-  out("fork",Left); out("fork",Right);
15         out("ticket").
16 +!thinking <- .my_name(Me); println(Me," thinking").
17 +!eat <- .my_name(Me); println(Me," eating").
```

Listing 1.1. The philosopher in JaCaMo [18].

Eating requires using the artifact for gaining forks and also a ticket that is used for avoiding deadlocks. Each agent implements the coordination policy in its plans, through the artifact operations *in* and *out*, as it directly manages

the acquisition and release of forks (and tickets). So, even though Agent-to-Agent decoupling is achieved, there is *no clear separation of concerns* between the agent programming and the artifact programming for what concerns the coordination logic; consequently, there is a tight coupling between the agents and the artifacts that allow their interaction. This hinders the specification of an agent programming methodology independent of that of the artifacts.

A first improvement to this solution (still in JaCaMo) could be placing the coordination logic inside the artifact; this could be achieved by: (1) moving the calls of *in* and *out* inside new higher level operations (in the following, *askForks* and *returnForks* respectively); and (2) introducing an observable property *availableForks* that is notified in the agent's belief base when forks are available:

```
+!eating: my_left_fork(Left) & my_right_fork(Right)
    <- askForks(Left, Right).
+availableForks(Left, Right) <- !eat; returnForks(Left, Right).
```

This second solution, though improved, is still not completely satisfactory. The relation between *askForks* and *availableForks* (the latter is a consequence of the former), that is fundamental to the programmer, is *hidden* inside the artifact. The agent invokes *askForks* (a service, at all respects) and, when forks are available, the artifact tells the agent through the observable property *availableForks*. Observable properties are *signals*, the agent is programmed to react to signals. Indeed, the plan corresponding to *availableForks* is activated as a reaction to the creation of the corresponding belief: The relation, that ties the plan to the event that activates it, is *causal*, but since it is not expressed explicitly, the agent is not enabled to perform any kind of reasoning. Also in this second solution, thus, such a relation depends on the coordination logic that is contained in the artifact. Once again, the lack of separation of concerns is troublesome for the specification of an agent programming methodology independent of artifacts.

```
/* Initial goals */
!counter(0).
/* Plans */
!start.
+!start: true
     <- focusWhenAvailable("philoArtifact");
        enact("philosopher").
+enacted(Id,"philosopher",Role_Id)
     <- +enactment_id(Role_Id);
        .my_name(Me);
        in("philo_init",Me,Left,Right);
        +my_left_fork(Left);
        +my_right_fork(Right);
        !!living.
+!living: counter(C)
     <- !thinking;
        !eating.
+!eating:my_left_fork(Left) & my_right_fork(Right) & counter(C)
    <- .my_name(Me); ?enactment_id(Role_Id);
       askForks(Left, Right, C).
+cc(My_Role_Id, "philosopher", available(Left,Right,C),
      returnForks(Left,Right,C),"DETACHED")
    : enactment_id(My_Role_Id) & my_left_fork(Left) &
      my_right_fork(Right) & counter(C)
    <- !eat(Left, Right, C);
       returnForks(Left, Right, C).
+cc(My_Role_Id, "philosopher", available(Left,Right,C),
     returnForks(Left,Right,C), "SATISFIED")
```

```
29      :   enactment_id(My_Role_Id) & my_left_fork(Left)
30    <-  ?counter(C); -+counter(C+1); !living.
31 +!eat(Left, Right, C): my_left_fork(Left) & my_right_fork(Right)
32         & available(Left, Right, C) & counter(C)
33    <-  .my_name(Me); ?enactment_id(Role_Id).
34        println(Me, " ", Role_Id, " eating").
35 +!thinking: counter(C)
36     <- .my_name(Me); ?enactment_id(Role_Id);
37        println(Me, " ", Role_Id, " thinking, time ",C).
```

Listing 1.2. The philosopher agent program in JaCaMo+.

In JaCaMo+, instead (Listing 1.2), coordination relies on social engagement. The agent program is built by exploiting the CoSE methodology. As in the JaCaMo solution, we suppose our agent has a !*living* main cycle (ln. 15) that alternates the goals !*thinking* and !*eating*. Coordination is needed just for eating: to this aim, forks must be available. Hence, we consider an environment that contains at least one artifact which satisfies one of the cases of step (2) in the methodology. Suppose case (a) is satisfied by an artifact exposing a role *philosopher*, which is empowered with an operation *askForks*, that will let the agent on standby until forks are available and, then, will create a commitment to return the assigned forks. The agent who executes the operation is the debtor of such a commitment, any other philosopher is the creditor. The antecedent condition is that forks are available and the consequent is that forks will be returned. Note that fork assignment is decided by way of a coordination policy that is implemented in the artifact.

Now, we need to program the agent behavior in occurrence of the state changes of such a commitment; indeed, only state changes that are meaningful to the aims of the agent are to be tackled. In our case, only *Detached* and *Satisfied* are meaningful. When the commitment is detached, the agent eats and then executes *returnForks*, thus satisfying its commitment. When the commitment is satisfied, the agent can re-start its main cycle (!*living*).

In this case, *askForks* is *not a mere service* by the artifact; it creates a social engagement, whose debtor is the requesting philosopher, and the creditor is the whole class of *philosophers*. This is done by the commitment *C(My_Role_Id, "philosopher", available(Left,Right,C), returnForks(Left,Right,C))*. The agent is requested to include one or more behaviors for managing such a commitment and, in particular, for managing the case in which it is *Detached*. This is possible because *askForks* and the *event +cc(My_Role_Id , "philosopher" , available(Left,Right,C), returnForks(Left,Right,C), "DETACHED")*[1] are *tied by the social meaning* of the operation in an explicit way, and this information is available to the programmer, who does not need to know the coordination logic that is implemented inside the artifact. Knowing the social meanings of artifact operations is sufficient for coordinating with others correctly. The connection between the event "commitment detached" and the associated plan is not only causal, but rather the plan has the aim of satisfying the consequent condition of the commitment (*returnForks*). Once again, there is not need of knowing or using logics that are internal to the artifact. Thus, we achieve not only Agent-to-Agent de-

[1] Meaning that a belief of type *cc*(...) was added to the agent's belief base.

coupling, but also a real Agent-Logic-to-Coordination-Logic Decoupling. Social meanings are the key element that enables the definition of an agent programming methodology.

5 Discussion and Conclusions

CoSE extends environments by realizing Engeström's *activity systems*, rather than mere artifacts. Citing [13, page 31], activity systems extend the classical triadic model (subject, object, and mediating artifact) in that the outcome is no longer momentary (situational), but consists of *new, objectified meanings* and *relatively lasting patterns of interaction*. In our case, objectified meanings are supplied by reified commitments, and all interactions are driven by such meanings instead than by the events (signs or signals), that are "physically" executed by the agents. Reified commitments also specify expected behaviors in terms of *what* is to be achieved rather than *how*.

The introduction of a commitment-based shared semantics of events allows for the design of software that meets many *software engineering principles*. First of all, *abstraction*: a failure to separate *behavior* (i.e., what) from *implementation* (i.e., how) is a common cause of unnecessary coupling among the components in a system. The what-level is tackled, in our proposal, by working at the level of commitments, and this is the only information that matters for carrying on an interaction/coordination. An agent who takes on a commitment assumes the responsibility that something will occur. Now then, who will make it occur, and which steps will bring to the outcome, are left to the how-level of single agent programming. Such steps may depend on the context, accommodating emerging opportunities, or managing specific difficulties. In other words, it is generally not necessary to impose any strict causal chain in signal generation.

Indeed, CoSE facilitates a *separation of concerns* between agent programming and the programming of agent coordination: As social relationships abstract the actual events upon which agents should coordinate, programmers can consider the programming of agents and the programming of coordination artifacts as two distinct problems. Programmers can define an agent's behavior on the sole basis of the *semantics* of social relationships, rather than on low-level events. Focusing on artifacts, programmers focuses on how the occurrence of an event changes the states of a set of social relations. Since social relationships are more abstract than events, the decoupling brings along further beneficial properties like *modularity* and *reuse*.

The *generality* principle calls for the development of software that is free from unnatural restrictions and limitations. This is precisely what is achieved by an approach that focuses on what rather than on how and that relies on a declarative, rather than on a procedural, representation. The *incremental development* principle advocates the incremental realization of software; e.g., one case at a time. Our proposal meets this requirement in that agent software can be developed by tackling one commitment at a time, or even one commitment state

change at a time. A carefully planned incremental development process can also simplify the management of changes in requirements.

Concerning agent-based design, many proposals are found in the literature. Briefly, SODA [15] is an agent-oriented methodology for the analysis and design of agent-based systems, adopting a layering principle and a tabular representation. It focuses on inter-agent issues, like the engineering of societies and environment for MAS, and relies on a meta-model that includes both agents and artifacts. However, SODA does not foresee social engagements nor it provides an agent programming methodology. GAIA [24] is a methodology for developing a MAS as an organization, not for programming agents. The 2CL Methodology [5] is an extension of [12]. It supports the design of commitment-based business protocols that include temporal constraints, and allows the verification of properties. As such, it could be used for helping the realization of artifacts, although the methodology is general and not oriented to this specific abstraction.

Social engagements are at the basis also of organization-oriented programming, of which JaCaMo is a prominent example. Recently, [25] proposed a JaCaMo extension that introduces Interaction Components to encode–in an automaton-like shape–protocols, where transitions are associated with (undirected) *obligations*. Such protocols provide *guidelines* of how organizational goals should be achieved. However, organization-driven guidelines are but a kind of interaction. We claim that when guidelines are missing, interaction should be supported based on the fundamental notions of goal and engagement. So, our proposal complements [25], and more in general organizational and normative approaches [11,14,16,10], in this respect. Social commitments [19], differently from obligations, are taken by agents as a result of internal deliberative processes, and can be directly manipulated by the agents, In addition, [22] shows how goals and commitments are strongly interrelated. Commitments are, thus, evidence of the agents' capacity to take responsibilities autonomously. Citing Singh [20], an agent would become a debtor of a commitment based on the agent's own communications: either by directly saying something or having another agent communicate something in conjunction with a prior communication of the debtor. That is, there is a *causal path* from the establishment of a commitment to prior communications by the debtor of that commitment. Such causal relationships are at the heart of the CoSE methodology. By contrast, obligations can result from a deliberative process which is outside the agent; this is the case of the interaction component in [25]. For a detailed discussion of the differences between obligations and commitments see [3].

Consequently, our proposal differs deeply also from proposals like [8], which also account for a social dimension of the MAS. That work, for instance, presents the SOPL language, that allows including, in each of the agents' programs, states of affairs that the agent tolerates (even though they are not explicit goals of its own), and rules to reason about the other agents' mental states (social conditions). Social conditions comprise possible evolutions depending on how other agents behave, they can vary from agent to agent, and are used by the agent in the process of deciding how to act. Nevertheless, mental states are private to

the agents and the absence of a semantics of actions based on mutual agreement makes speculations about the others' behavior fragile, because expectations do not base upon explicit engagements. From a software engineering perspective, then, since evolutions of interactions are encoded in the very agent programs, the proposal does not support the decoupling of agents from the interaction logic.

Future work will concern tackling the formal notions of *social context* and of *enactment of a protocol in a social context* introduced in [4], as well as further exploring the use of typing systems, along the lines of [2].

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).

References

1. Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. A Commitment-based Infrastructure for Programming Socio-Technical Systems. *ACM Transactions on Internet Technology, Special Issue on Foundations of Social Computing*, 14(4):23:1–23:23, December 2014.
2. Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. Typing multi-agent systems via commitments. In *Engineering Multi-Agent Systems*, volume 8758 of *LNCS*, pages 388–405. Springer International Publishing, 2014.
3. Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati, and Roberto Micalizio. Leveraging Commitments and Goals in Agent Interaction. In D. Ancona, M. Maratea, and V. Mascardi, editors, *Proc. of XXX Italian Conference on Computational Logic, CILC*, 2015.
4. Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, and Munindar 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.
5. Matteo Baldoni, Cristina Baroglio, Elisa Marengo, Viviana Patti, and Federico Capuzzimati. Engineering commitment-based business protocols with the 2CL methodology. *JAAMAS*, 28(4):519–557, 2014.
6. Olivier Boissier, Rafael H. Bordini, Jomi F. Hbner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747 – 761, 2013.
7. Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons, 2007.
8. Francesco Buccafurri and Gianluca Caminiti. Logic programming with social features. *Theory and Practice of Logic Programming (TPLP)*, 8(5-6):643–690, 2008.
9. Rosaria Conte, Cristiano Castelfranchi, and Frank Dignum. Autonomous Norm Acceptance. In *ATAL*, pages 99–112, 1998.
10. Natalia Criado, Estefania Argente, Pablo Noriega, and vicent Botti. Reasoning about norms under uncertainty in dynamic environments. *International Journal of Approximate Reasoning*, 2014.

11. Mehdi Dastani, Davide Grossi, John-Jules Ch. Meyer, and Nick A. M. Tinnemeier. Normative multi-agent programs and their logics. In *Normative Multi-Agent Systems, 15.03. - 20.03.2009*, volume 09121 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.
12. Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Trans. Softw. Eng. Methodol.*, 19(2), 2009.
13. Yrjö Engeström, Reijo Miettinen, and Raija-Leena Punamäki, editors. *Perspectives on Activity Theory*. Cambridge University Press, Cambridge, UK, 1999.
14. Felipe Meneguzzi and Michael Luck. Norm-based behaviour modification in BDI agents. In *AAMAS (1)*, pages 177–184. IFAAMAS, 2009.
15. Ambra Molesini, Andrea Omicini, Enrico Denti, and Alessandro Ricci. SODA: A roadmap to artefacts. In *Engineering Societies in the Agents World VI*, volume 3963 of *LNAI*, pages 49–62. Springer, 2006. 6th Int. Workshop (ESAW 2005).
16. Daniel Okouya, Nicoletta Fornara, and Marco Colombetti. Proc. of EMAS. In *EMAS@AAMAS*, volume 8245 of *LNCS*, pages 215–234. Springer, 2013.
17. Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *JAAMAS*, 17(3):432–456, 2008.
18. Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.
19. Munindar P. Singh. An ontology for commitments in multiagent systems. *Artif. Intell. Law*, 7(1):97–113, 1999.
20. Munindar P. Singh. Commitments in multiagent systems some controversies, some prospects. In Fabio Paglieri, Luca Tummolini, Rino Falcone, and Maria Miceli, editors, *The Goals of Cognition. Essays in Honor of Cristiano Castelfranchi*, chapter 31, pages 601–626. College Publications, London, 2011.
21. Pankaj R. Telang, Munindar P. Singh, and Neil Yorke-Smith. Relating Goal and Commitment Semantics. In *Post-proc. of ProMAS*, volume 7217 of *LNCS*. Springer, 2011.
22. Pankaj R. Telang, Neil Yorke-Smith, and Munindar P. Singh. Relating Goal and Commitment Semantics. In *Proc. of ProMAS*, volume 7212 of *LNCS*, pages 22–37. Springer, 2012.
23. Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *JAAMAS*, 14(1):5–30, 2007.
24. Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Trans. Softw. Eng. Methodol.*, 12(3):317–370, 2003.
25. Maicon R. Zatelli and Jomi F. Hübner. The Interaction as an Integration Component for the JaCaMo Platform. In *Proc. of EMAS*, 2014.