# Agent Protocols for Social Computation

Michael Rovatsos, Dimitrios Diochnos, and Matei Craciun

School of Informatics
The University of Edinburgh
Edinburgh EH8 9AB, United Kingdom
{mrovatso,ddiochno,s1374265}@inf.ed.ac.uk

**Abstract.** Despite the fact that social computation systems involve interaction mechanisms that closely resemble well-known models of agent coordination, current applications in this area make little or no use of the techniques the agent-based systems literature has to offer. In order to bridge this gap, this paper proposes a data-driven method for defining and deploying agent interaction protocols that is entirely based on using the standard architecture of the World Wide Web. This obviates the need of bespoke message passing mechanisms and agent platforms, thereby facilitating the use of agent coordination principles in standard Web-based applications. We describe a prototypical implementation of the architecture and experimental results that prove it can deliver the scalability and robustness required of modern social computation applications while maintaining the expressiveness and versatility of agent interaction protocols.

**Keywords:** Agent Communication, Social Computation, Web Agents

## 1 Introduction

Most real-world *social computation* applications that involve large-scale human and machine collaboration (e.g. collective intelligence [11] or human computation [9]), are currently implemented using either *ad hoc* methods or programming frameworks [1, 10] that make no use of agent technology. Within the agents community, on the other hand, agent communication languages and interaction protocols [3] have been widely used to design and deploy a wide range of agent coordination mechanisms, many of which bear close similarity to those needed in social computation systems. This is at least in part due to the fact that the *architectural* proposals for developing real-world agent-based systems mostly rely on bespoke platforms with custom message passing mechanisms and control structures. Since the inception of those agent platforms, the architecture of the Web [6] has given rise to a plethora of massive-scale distributed applications, almost in complete ignorance of agent-based techniques [14].

The work presented in this paper aims to bridge the gap between agent coordination techniques and social computation by providing a method for mapping the principles of agent protocol design to the architecture of the Web. We describe a data-driven method for defining and deploying agent interaction protocols that complies with the architecture of the Web, and does away with a need for point-to-point messaging infrastructures. Also, contrary to many existing agent platforms, it does not assume ideal conditions regarding liveness of agent processes and availability of perfect communication channels.

The basic principles of this architecture are simple: It conceives of messages as entries in persistent data stores accessible via normal HTTP operations, and models dependencies between these messages through an explicit graph structure, where causal and temporal links between messages in a protocol are exposed to agents via Web APIs. This enables avoiding redundant messaging in broadcast situations, failure recovery and management of "stale" interactions, lightweight *ex post* modification of previous interactions, as well as global monitoring and analysis of coordination processes. Also, it leverages the architecture of the Web to enable lightweight communication and is oblivious to the degree of centralisation applied in systems design. We describe a prototypical implementation of our architecture in a typical application scenario that allows us to demonstrate its benefits. Our experiments with a deployed prototype show that our approach offers significant advantages in terms of scalability and robustness.

The remainder of the paper is structured as follows: We start by introducing an example scenario in section 2 that serves to illustrate our framework, and is also later used in our experiments. Section 3 introduces our formal framework for modelling conventional agent interaction protocols and their semantics. Our data-driven architecture is presented in section 4 together with a discussion of its properties. Experiments are presented in section 5, and after reviewing related work in section 6, section 7 concludes.

## 2 Example

A typical social computation scenario that involves large-scale agent collectives, and which we will use for illustration purposes throughout the paper, is ridesharing (see, for example, `blablacar.com` and `liftshare.com`), where travellers (drivers and passengers) request rides posting location, price, and possibly other constraints. Ridesharing is a representative example both due to the range of functions it requires (matchmaking, negotiation, teamwork) and because it exhibits many characteristics of real-world collective collaboration systems (many users, asynchronous communication, heterogeneity of user platforms).

The *team task* protocol shown in figure 1 describes a possible coordination mechanism that could be used in such a system, following a traditional agent-based model which involves an orchestrator $o$ and task peers $p$ in an 1:n relationship. In the top section of the diagram, peers `ADVERTISE` their capability to play role $r$ in action $a$, e.g. driving a car, occupying a passenger seat, or paying a driver in the case of ridesharing. This advertisement is acknowledged simply to terminate this stage with a definite response. In the subequent *matchmaking* stage, peers may `REQUEST` a task, i.e. a plan that will achieve getting from initial state $I$ to goal state $G$, subject to certain constraints $C$ (e.g. a price limit).

Based on requests from various agents and by using the capabilities they have advertised, $o$ proposes a possible task $t$ that would involve a specific plan $\pi$ to be executed, and a role assignment for the participants clarifying which agent has to perform which actions, or tells $p$ that `NO_SOLUTIONS` can be found. In the case of ridesharing, the plan would be the ride specification, quoting a price, time, and possibly other constraints (e.g. whether smoking is allowed). If a peer `AGREE`s to a task, this might have become invalid in the meantime because others have `REJECT`ed it. If the task is still valid, and once all participants agree, the orchestrator invites participants to `START` executing the plan, after which $p$ can `UPDATE` the execution status $st$ of individual steps $a_i$ in the plan (e.g. "we reached the destination") or provide feedback reports $F$ regarding the task (e.g.

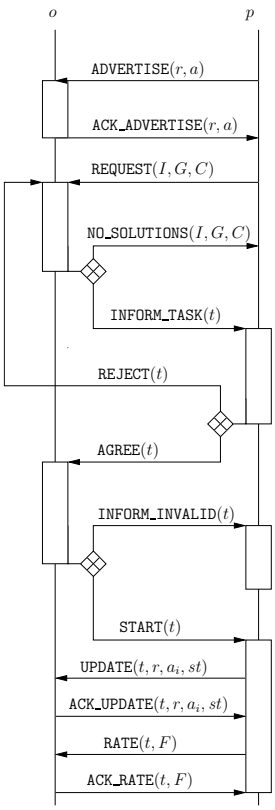



**Fig. 1.** The team task protocol

"the driver was driving too fast"). These steps can be repeated depending on how many steps there are in $t$ in the case of `UPDATE`, or without limitation in the case of `RATE`. If $p$ rejects a task, more tasks might be suggested until no more solutions exist. Note that the protocol deliberately contains a few "imperfections": First of all, once a peer agrees, she will not be notified if other participants reject the task she agreed too. Secondly, there is no timeout for the negotiation. Hence, a peer will never be told that the negotiation failed because some participants did not respond. Below, we will explain how our proposed architecture results helps address such issues without complete protocol re-design.

## 3 Formal framework

Before proposing our own approach, we introduce a formal framework that allows us to capture action and communication semantics in a decentralised agent-based system. Our formalism is based on a plan-based semantics of communication and action, i.e. we consider a state transition system where messages and actions (we use this term to denote non-communicative actions that change the environment state) modify the values of state variables. While this follows traditional STRIPS-style planning formalisms [7], note we are not assuming the use of actual planning algorithms in the system. The notation just gives us a simple, generic way of describing a discrete, distributed state transition system.

Let $V = \{v_1, v_2, \ldots, v_k\}$ variables that range over a domain of discourse $D$[1], and *constraints* $c = \{(v_1, D_1), \ldots, (v_m, D_m)\}$ denoting that $v_i \in D_i \subseteq D$, where all $v_i$ are distinct, and $\{v_1, \ldots, v_m\} \subseteq V$ is called the *domain* $dom(c)$ of constraint $c$. We call a constraint *failed* if any for any $v_i \in dom(c)$ we have $D_i = \emptyset$, and write $c = \bot$ in this case. A *substitution* $\theta$ is a constraint $\{(v_1, E_1), \ldots, (v_l, E_l)\}$ that can be applied to $c$ to result in a new constraint $c\theta = \{(v_1, D'_1), \ldots, (v_m, D'_m)\}$ such that for all $v_i = v_j$ with $v_i \in dom(c)$ and $v_j \in dom(\theta)$ we have $D'_i = D_i \cap E_j$. A substitution is called a *grounding* if $|D'_1| = |D'_2| = \ldots = |D'_m| = 1$. A grounding is *admissible* if $c\theta \neq \bot$, and we write $\lfloor c \rfloor$ for the set of all possible *groundings* or *instances* of $c$. Entailment among two constraints is defined as $c \models c'$ if $\lfloor c \rfloor \subseteq \lfloor c' \rfloor$, i.e. $c$ is a stricter constraint satisfied by some groundings of $c'$.

Next, we introduce agents and their actions. Assume *agents* $A = \{a_1, a_2, \ldots a_n\}$, and, in slight abuse of notation, let their names also be valid variable values, i.e. $A \subseteq D$. We consider a timed system with execution steps $T = \{t_1, t_2, \ldots\}$ using a global clock shared by all agents. For any variable $v \in V$, $a_i$ may have a local copy whose value can change over time. We write $v_i^j$ for the value of $v$ for agent $i$ at time step $j$ ($V_i$ denotes agent $i$'s local variables, and $V_i^j$ their values at time $j$). We may drop the subscript and write "$v = d$" for some variables whenever all agents' local copies agree on the variable value, i.e. $v = d \Leftrightarrow \forall i\ .v_i = d$. *Fluents* $F = \{f_1, \ldots, f_k\} \subseteq V$ are variables that describe system states, and exclude any auxiliary variables not used to reflect system state, e.g. the roles denoting senders and receivers of messages in message schemata (see below). A *state specification* $S$ is a constraint with $dom(S) \subseteq F$, and is used to represent the set of all states $s \in \mathcal{S}$ with $s \models S$. A state $s$ can be viewed as a constraint that is a full variable assignment $\{(f_1, \{d_1\}), \ldots, (f_k, \{d_k\})\}$ for all domain fluents in $F$. When referring to states, we will write $s_i^j$ to denote a full assignment to concrete values for agent $i$ at time $j$.

An *action* $ac = \langle \{a_1, \ldots, a_k\}, pre, \mathit{eff} \rangle$ is performed by agents $\{a_1, \ldots, a_k\} \subseteq A$ and is associated with two constraints, its *preconditions* $pre$ and *effects* $\mathit{eff}$ with $dom(pre) \cup dom(\mathit{eff}) \subseteq F$. For any $s \in \mathcal{S}$ with $s \models pre(ac)$ ($ac$ is *applicable* in $s$), execution of $ac$ results in a *successor state* $succ(s, ac) = s'$ where $s' = s \backslash \{(v, D) | v \in dom(\mathit{eff}(ac))\} \cup \mathit{eff}(ac)$. In other words, if $ac$ is applicable in $s$, then the successor state results from removing the values of all *affected* variables in $dom(\mathit{eff}(ac))$ from $s$ and adding their new values as per $\mathit{eff}(ac)$. Note that these actions need not be "ground" in the sense that the fluents they involve need to have specific single values before or after the action. A *plan* $\pi = \langle ac_1, \ldots, ac_n \rangle$ is a sequence of actions such that $ac_1$ is applicable in the states represented by the initial state specification $I$, i.e. $I \models pre(ac_1)$, and each $ac_i$ is applicable in $succ(s, \langle ac_1, ac_2, \ldots, ac_{i-1} \rangle)$ (where $succ$ is canonically extended to sequences of actions), for all $2 \leq i \leq n$ and $s \models I$. Plans provide the definition for any well-defined sequence of actions that is feasible given the specifications of these actions and the current system state. A plan $\pi$ is a *solution* for a planning problem $\langle I, G, Ac \rangle$ with initial state specification $I$ and goal state specification $G$ if $succ(s, \pi) \models G$ for all $s \models I$, i.e. if its execution from any state that satisfies $I$ results in a state that satisfies $G$.

Given this general framework, we can proceed to defining the structure and semantics of agent protocols.

[1] Different types $D_j$ can be used here to accommodate different types of variables. These are omitted for simplicity.

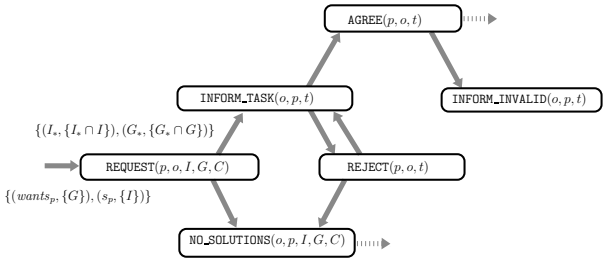
**Definition 1** *A* message schema $\mu = \texttt{MSG}(se, re, c, pre, eff)$ *is a structure with label* `MSG`*, where* $se$ *and* $re$ *are variables for the sender(s) and receiver(s) of the message, constraint* $c$ *denotes the message content, and precondition/effect constraints* $pre$ *and* $eff$ *with* $dom(pre) \subseteq V$ *and* $dom(eff) \subseteq V$*.*

In figure 1, such schemata label the edges connecting the individual boxes on the swimlanes of the diagram (which represent sender $p$ and receiver $o$), e.g. $\texttt{REQUEST}(p, o, \{(i_p, \{I\}), (g_p, \{G\}), (c_p, \{C\})\})$. For readability, we omit preconditions and effects and the constraint notation assigning concrete values to $p$'s local variables for $I$, $G$, and $C$ is not used in the diagram (the request implies, for example, that $p$'s local variable $i_p$ has value $I$ at the time of sending).

To define the structure of a protocol, we introduce a graph that is "dual" to that in the diagram, in that is has message schemata for nodes and edges for decision points:

**Definition 2** *A* protocol graph *is a directed graph* $P = \langle \Phi, \Delta \rangle$ *whose node set includes a set of message schemata uniquely identified by message labels, with additional root and sink nodes* $start$ *and* $end$ *(*$eff(start) = pre(end) = \emptyset$*). Its edges are given by a mapping* $\Delta : \Phi \to 2^{\Phi}$*. Every edge* $(\mu, \mu')$ *with* $\mu' \in \Delta(\mu)$ *is labelled with* $eff(\mu)$ *and* $pre(\mu')$*.*[2]

We present the protocol graph for part of the team task protocol (preconditions end effects are only shown for `REQUEST`):




The example assumes that the precondition for `REQUEST` is $\{(wants_p, G), (s_p, I)\}$ for $p$. The effect for $o$, who is gathering planning problems from peers to propose a joint plan that solves all of them, is $\{(I_*, I_* \cap I), (G_*, G_* \cap G)\}$ where "$_*$" denotes the view $o$ has of all peers. In other words, $o$'s strategy involves conjunctively narrowing down initial and goal states before suggesting a plan that satisfies all of them. To make the plan-based semantics of protocols concrete, we need to introduce messages as instances of schemata:

**Definition 3** *A* message *is a structure* $m = \langle \mu, Se, Re, \theta, t \rangle$ *where* $\mu$ *is a message schema,* $Se \subseteq A$ *and* $Re \subseteq A$ *are the (non-empty) sets of senders*[3]*/receivers of the message,* $\theta$ *is a substitution for* $c(\mu)$*, and* $t$ *the time the message was sent.*

---

[2] Throughout the paper, we adopt the convention of referring to elements in a structure $x = \langle y, y', \ldots \rangle$ as $y(x)$, $y'(x)$ etc.

[3] Allowing many senders in messages may seem counter-intuitive at first, but is useful for situations where a physical sender acts on behalf of a whole group, or to summarise identical messages received from various peers as one message in the data-driven model we introduce in section 4.

The following definition defines when a message is *admissible*, i.e. it is a legal continuation of an observed interaction:

**Definition 4** *For any protocol graph $P$, state $s^t \in \mathcal{S}$, and initial message sequence $\pi = \langle start, m_1, \ldots, m_{t-1} \rangle$, define:*

$$\begin{aligned}\langle \pi, s_t \rangle \models_P m_t \quad :\Leftrightarrow \quad & m_t = end \wedge end \in \Delta(\mu(m_{t-1})) \quad \vee \\ \big(\theta = & \theta(m_1)\theta(m_2)\cdots\theta(m_t) \neq \bot \wedge \mu(m_t) \in \Delta(\mu(m_{t-1})) \wedge \\ & \forall i \in Se(m_t).\ s_i^t \models pre(\mu(m_t)\theta) \wedge \\ & \forall j \in Re(m_t).\ s_j^{t+1} \models succ(s_j^t, \mu(m_t)\theta)\big) \qquad \square\end{aligned}$$

This defines a message $m_t$ as admissible in the context of a current message sequence and state $\langle \pi, s^t \rangle$, which we call a *model* for $m$, if either $m_t = end$ and its immediate predecessor was connected to the *end* node in $P$, or if (i) its schema $\mu(m_t)$ is a successor to that of the most recent message, (ii) the preconditions (effects) of that schema are satisfied by all senders (receivers) of the message in timestep $t$ $(t+1)$, and (iii) this is subject to the combined substitution $\theta$ that accumulates all the substitutions applied in previous messages (and which must itself be consistent).[4]

In other words, an admissible message is interpreted as a planning action $\langle Se \cup Re, pre(\mu(m)\theta), \mathit{eff}(\mu(m)\theta) \rangle$, with the additional requirement that it extends the observed message sequence following $P$, and respects the substitions applied to earlier messages on the path that led to it.

To extend this definition to message sequences, we can write $\langle \pi, s \rangle \models_P \pi'$ for any finite $\pi' = \langle m_{t+1}, \ldots, m_{t+k} \rangle$ iff

$$\big\langle \pi\langle m_{t+1}, \ldots, m_{t+j} \rangle, succ(s, \langle m_{t+1}, \ldots, m_{t+j} \rangle) \big\rangle \models_P m_{t+j+1}$$

for all $0 \leq j \leq k-1$. We write $s \models_P \pi$) iff $\langle \langle \rangle, s \rangle \models_P \pi$.

With this, we can proceed to define the semantics of a protocol through the set of admissible *continuations* it gives rise to in a specific state given an observed execution history:

**Definition 5** *Let $s \in \mathcal{S}$ and $\pi$ a message sequence. If $s \models_P \pi$, the* continuations *$[\![\pi]\!]_s$ of $\pi$ are defined as the (potentially infinite) set of sequences messages $\pi'$ for which $\langle \pi, s \rangle \models \pi'$ holds. We let $[\![\pi]\!]_s := \bot$ if $s \not\models_P \pi$.* $\square$

This completes our account of a simple and fairly generic plan-based semantics for agent interaction protocols. Our semantics does not make any specific commitment as to the actual semantic language (e.g. mentalistic, commitment-based, or deontic) used to specify constraints governing the exchange of messages. Instead, it specifies what message sequences are admissible under a shared protocol definition, and how message passing results in a synchronisation among agents' local variables. For simplicity, we have assumed that no additional agent actions or exogenous events occur during protocol execution. Note, however, that such

[4] Note that different semantics are possible here, which may assume that senders also have a modified state regarding their perception of receivers' local variables after sending a message, or receivers inferring facts about senders' previous states upon receipt of a message. Which of these variants is chosen is not essential for the material provided below.

actions or events could be easily accommodated in the protocol graph as additional choices between successive messages without requiring additional formal machinery.

# 4 Data-centric Architecture

## 4.1 Framework

While conventional specifications of agent interaction protocols such as the ones considered above provide a very flexible framework for coordinating multiple agents, the point-to-point message passing they assume can be problematic in large-scale multiagent systems using potentially unreliable communication infrastructures, and operating over long periods of time, so that the contributions of agents occur at unpredictable points in time.

Consider a real-world deployment of the protocol shown in figure 1 in a web-based ridesharing application with many users. If we use conventional message passing, this protocol would require $n$ conversations for $n$ task peers going on in parallel, and $o$ would need to maintain separate internal data structures to track which agents have already agreed to the task, which of them may provide execution updates, etc. Also, these conversations would have to remain "open" indefinitely, unless strict time limits were imposed on these parts of the protocol. Another drawback is that many data objects such as identical requests, suggested tasks, or information about invalid/agreed tasks and initiation of task execution would have to be sent repeatedly from/to different peers. Finally, if we wanted to de-couple different parts of the protocol that are not causally linked to each other in order to allow for a more flexible execution of the different stages of the protocol (e.g. advertising capabilities is unrelated to negotiation), this would involve creating separate protocols, and managing synchronisation among variables that are not local to a single protocol.

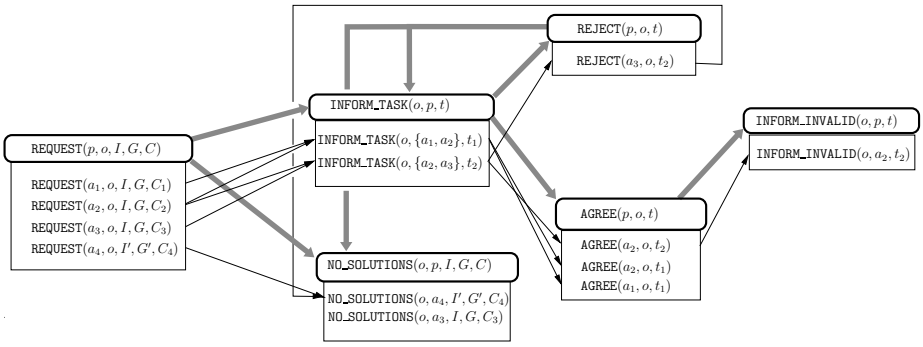



**Fig. 2.** Data-centric model of part of the ridesharing protocol

Before introducing our data-driven architecture to address some of these issues, we present its instantiation for the negotiation part of our team task protocol as an example in figure 2. The diagram combines the original protocol graph (message schemata in rounded boxes, connected with bold grey arrows) with *message stores* attached to every schema. These message stores contain

messages exchanged by the participants so far, and links (black arrows) between messages that were generated in response to each other. As before, we omit preconditions and effects as well as timestep labels and the details of content constraints for readability.

Using linked message stores enables us to replace message passing among agents by inspecting and modifying the contents of persistent message repositories, which is the key idea behind our approach. We start by introducing *protocol execution graphs (PEGs)*, which provide the link structure arising from observed message sequences:

**Definition 6** *Let* $\Pi = \{\pi_1, \ldots, \pi_k\}$ *a set of protocol executions where* $\pi_i = \langle m_1, \ldots, m_j, \ldots, m_{t_i} \rangle$ *and* $\pi_{ij} = m_j$*, and* $M(\Pi) = \{\pi_{ij} | \pi_i \in \Pi\}$ *the set of all messages in* $\Pi$*. The* PEG *is a directed graph* $P(\Pi) = \langle M(\Pi), \Delta(\Pi) \rangle$ *with edges* $\Delta(\Pi) = \{(m, m') | \exists \pi_i \in \Pi . m = \pi_{ij} \wedge m' = \pi_{ij+1}\}$.

*For any set of messages, we define a mapping* $\varphi : M \to \Phi(P)$ *to the nodes in* $P$*, where* $\varphi(m) = \mu$ *if* $\exists \mu \in \Phi(P) . \mu = \mu(m)$ *and* $\perp$ *else. Given this,* $\pi_i \in \Pi$ *is associated with a* generating path $\varphi(\pi_i) := \langle \varphi(m_1), \ldots, \varphi(m_{t_i}) \rangle$ *in* $P$. □

A PEG has every two messages connected that correspond to message schemata connected in the protocol graph the executions followed. Note that whenever the protocol graph contains cycles, a PEG may contain unfoldings of these cycles (and thus message schemata may appear repeatedly in a generating path $\varphi(\pi_i)$). Furthermore, even though the distinct message schema labels guarantee that every message has a unique node in the protocol graph assigned to it, identical messages (sent to or from different agents) appear only once in the graph. On the other hand, if two messages have identical senders, receivers, and conent, they would count as different nodes in the PEG if they were part of different conversations (as they are annotated with different timestamps). In figure 2, the nodes of the PEG are the entries of the boxes under each message schema, and its edges are depicted as black arrows connecting these nodes.

As concerns continuations, we can extend our previous definitions canonically to sets by letting $\langle \Pi, s \rangle \models_P m$ iff $\exists \pi \in \Pi . \langle \pi, s \rangle \models_P m$ and $[\![\Pi]\!]_s := \cup_{\pi \in \Pi} [\![\pi]\!]_s$.

The final step in our construction is to identify *message stores*, one for each message schema $\mu$ appearing in the protocol graph (shown as square boxes in figure 2). These provide a somewhat orthogonal view of the PEG, focusing on specific message schemata:

**Definition 7** *A* message store *is a set of messages* $M_\mu := \{m \in M(\Pi) | \mu(m) = \mu\}$ *containing all message instances for a message schema* $\mu$*. It supports the following operations given* $m = \langle \mu, Se, Re, \theta, t \rangle$*:*

- $get(a, M_\mu) = \{m \in M_\mu | a \in Re(m)\}$
- $add(a, M_\mu, m) = M_\mu \Leftrightarrow a \in Se(m) \wedge M'_\mu = M_\mu \cup \{m\}$
- $del(a, M_\mu, m) = M'_\mu \Leftrightarrow a \in Se(m) \wedge M'_\mu = M_\mu \backslash \{m\}$
- $mod(a, M_\mu, m, m') = add(a, del(a, M_\mu, m), m')$

*The operations add, del (and mod) leave* $M_\mu$ *unchanged if their arguments do not satisfy the above constraints.* □

The main reason we define message stores as first-order citizens in our architecture is that they permit the definition of operations which can be used to emulate sending and receiving messages. These operations, which are realised

as physical messages over the network (but we distinguish from protocol messages) allow an agent to create a new message if it is a sender of that message (*aff*), and to inspect those messages in a store that have her as receiver (*get*). We also permit deletion of previous messages through *del* for reasons that will become clear below, and modification of an existing message through *mod* (a combination of *del* and *add*).

Using these methods, a message such as $\texttt{REQUEST}(a_1, o, I, G, C_2)$ in figure 2 would be realised as a sequence of calls $add(a_1, M_\mu, \texttt{REQUEST}(a_1, o, I, G, C_2)) \rightarrow get(o, M_\mu)$ where $M_\mu$ is the message store for `REQUEST`. This enables a different way of processing the protocol specification, which is based on an ability to generate responses to any message contained in a message store without requiring a control flow that manages every conversation sequence individually:

**Proposition 1** *Let $s^t \in \mathcal{S}$ and $m = \langle \mu, Se, Re, \theta, t \rangle$ a message with $\langle \Pi, s^t \rangle \models_P m$. We define*

$$op = get(Re, add(a, M(\mu), m)))$$

*where $a \in Se$ and $get(Re, \ldots)$ is shorthand notation for all receivers executing the get operation in any ordering. We assume that each get/add operation takes one timestep. Further, we assume that the add operation is only performed if $s_i^t \models pre(\mu\theta)$ for all $a_i \in Se$, and all $a_j \in Re$ update their local state $s_j^t$ to $s_j^{t+1} = succ(s_j^t, \mu\theta)$ instantly when they observe any new message $m$.*

*Then, if $|Re| = k$, and no other actions or message store operations are executed between $t$ and $t+k$, it holds that $M'_\mu = op(M) = M \cup \{m\}$ in $s^{t+k}$ and $succ^{(k)}(s^t, op) = succ(s^t, m)$.*[5]

Proof. The proof for this proposition requires only straightforward application of the respective definitions. The operation *op* on $M_\mu$ involves one sender *add*ing $m$ to the message store (which implies $M'_\mu = op(M) = M \cup \{m\}$), and $k$ receivers *Re get*ting the result. Since the message is admissible, we would have $s_i^t \models pre(\mu\theta)$ for all $i \in Se$ and $s_j^{t+1} \models succ(s^t, \mu\theta)$ if this message was sent. We assume that the *get* message is only sent if the sender can locally satisfy the preconditions of $m$, and that receivers incorporate the effects of any new message observed on a message store locally (though for a given agent this will only happen at $s^{t+l}$ for some $1 \leq l < k$ depending on when the receiver performs the *get* operation). Given this, and under the assumption that no other action occurs while *op* is being executed, we have $succ^{(k)}(s^t, op) = succ(s^t, m)$. ■

The importance of this proposition is twofold: Firstly, it shows how message store operations can correctly replace any protocol message exchange. Secondly, it reveals that an additional $|Re|$ *get* operations are necessary to produce the same outcome, and that the receivers monitor the contents of each relevant message store continually. On the other hand, it is sufficient if the time $k$ required for these updates is less than the time that passes until further messages being sent to or from the recipients, or other actions are executed that affect their local state. Our model also allows for more unusual operations on message stores, for example deletions of past messages. While this might seem counterintuitive, we discuss in section 4.2 how it can be very useful in real applications. Deletions

[5] The superscript $(k)$ is added to the *succ* function here to indicate that *op* requires this number of timesteps.

require a more complex "rollback", which obviously cannot undo the global state of the system, but for which we can establish a weaker result:

**Proposition 2** *For any message $m$, let $next(m, \Pi) = \{m' | (m, m') \in \Delta(\Pi)\}$ with $next^*(m, \Pi)$ as its reflexive and transitive closure. Removing $next^*(m, \Pi)$ results in a PEG $\Pi' = \langle \Phi', \Delta' \rangle$ where $\Phi' = \Pi \backslash next^*(m, \Pi)$ and $\Delta' = \Delta(\Pi) \backslash \{(m', m'') | \{m', m''\} \cap next^*(m, \Pi) \neq \emptyset\}$. It holds that:*

1. *If $\forall m' \in next^*(m, \Pi)$. $M'_{\mu(m')} = del(a, M_{\mu(m')}, m_i)$ and $\forall m' \notin next^*(m, \Pi)$. $M'_{\mu(m')} = M_\mu(m')$, then $M'_\mu = M_\mu(\Pi')$ for all $\mu$ and some $a \in A$.*
2. *For any $m \in \Pi'$ we have $\langle \Pi^{t(m)}, s^{t(m)} \rangle \models_P m_j$ where $\Pi^{t(m)}$ and $s^{t(m)}$ are the contents of the original PEG and state at time $t(m)$ when the message was created.*

Proof. Statement 1. claims that deleting all successors of $m$ from the respective message stores, and leaving all other message stores unchanged will restore the property that any message store $M_\mu$ in the system contains all messages instantiating a schema $\mu$ in $\Pi'$. This is trivially the case, as $\Pi'$ is identical to $\Pi$ with the exception of having $m$ and all its successors and their adjacent edges removed. To see that stement 2. holds, it suffices to observe that all remaining messages in $\Pi'$ are either a predecessor of $m$, or occur on paths that do not contain $m$ or any of its successors. It follows that their validity at the time of their creation is maintained if we remove $m$ and all subsequent messages. ■

The main implication of this result is that when a message is deleted from a store, then all its successors need to be deleted with it to maintain some level of consistency (this also assumes that no other messages or modifications on message stores take place in the meantime). Even with these provisions, the level of consistency achieved is obviously much weaker than what can be guaranteed for *add* operations, as deletions remove paths that were previously available, and only paths unaffected by the removal of $m$ have identical continuations as before the removal. Also, the system state may have changed compared to when the original messages were sent, so that we may not be able to track what interactions brought it about. Finally, we should note that the two properties we have just established apply to *mod* operations as a consequence of those operations being abbreviations for a composition of *del* and *add* calls.

### 4.2 Discussion

To illustrate the use of our model, let us revisit the example from figure 2 in more detail: We have six initial `REQUEST` messages from agents $a_1, \ldots a_4$, which result in possible tasks $t_1$ for $\{a_1, a_2\}$, $t_2$ for $\{a_2, a_3\}$ and no solution for $a_4$ (maybe because his requirements don't match those of any other peers). One immediately obvious advantage of our approach here is that only `INFORM_TASK` messages need to be "sent" to two agents each (sets $\{a_1, a_2\}$ and $\{a_2, a_3\}$).

Next, we have the situation that $a_2$ `AGREE`s to $t_2$ and $a_3$ `REJECT`s this task. We assume that $a_2$ is the driver and needs to agree first (no ride can be taken without a car), but there is no such restriction regarding rejection, which any participant of the task can issue at any point. Now if $a_3$ issues the rejection first, $a_2$ will receive an `INFORM_INVALID` response, as shown in the diagram, and no agreement on $t_2$ will be possible anymore. If $a_2$ has already agreed, however, this agent will never be notified of $a_3$'s rejection, a problem we already mentioned in section 2. One solution to this problem would be to add an edge

from INFORM_INVALID to AGREE, which was not included in the original protocol of figure 1. Since previous INFORM_TASK messages also gave $a_2$ the option $t_1$, she can now agree to this task, and after $a_1$ agrees, too, the next message would be START to initiate task execution.

This is generally how protocol flexibility has to be accommodated in normal agent protocols – every possible agent behaviour has to be accounted for by providing additional paths that enable other agents to respond appropriately. In fact, the INFORM_INVALID→AGREE edge would not work here, as no alternative possible task $t_1$ would be known to $a_2$ (unless a list of all possible tasks was sent with INFORM_TASK from the outset, which would doubtlessly complicate the workflow further). So, we would have to backtrack at least to the level of INFORM_TASK (in a "task no longer available, here's another alternative" fashion) to allow $a_2$ to make alternative choices. Or we would leave $a_2$'s AGREE message without response, whereupon we would rely on the decision logic of the agent to resolve the problem (e.g. by having her assume failure after some time).

Our data-centric view affords us with additional ways of dealing with such problems. Firstly, because of our protocol semantics, the INFORM_INVALID option is easy to accommodate, as $a_2$ can still AGREE to any task contained in the INFORM_TASK message store. Secondly, the orchestrator could remove INFORM_TASK $(o, \{a_2, a_3\}, t_2)$ (as owner of this message) after receiving a REJECT from $a_3$, and $a_2$ would be able to anticipate that its previous AGREE message has become invalid (it could even be *del*eted by $o$ if we used this type of call and gave the orchestrator appropriate permissions for this operation on messages not created by herself). Under these circumstances, not even the INFORM_INVALID message itself would be necessary, thus making the protocol even simpler. Finally, we could give $a_3$ permission to *add* INFORM_TASK messages (for example with possible alternative tasks that were not generated by $o$) or post *mod*ifications to $t_2$ in order to make the task acceptable for her, thus increasing the chances that successful agreement would be reached.

Thus, even though in principle the possible computations that can be jointly performed by agents are of course no different from the agent-centric view, our data-centric view allows much more flexibility in organising the interactions that lead to those computations, without requiring that the overall protocol needs to be significantly re-designed to accommodate additional functionality. For example, we could have orchestrators post arbitrary new tasks in an asynchronous way (for the same requests, or incrementally, as more potentially matching agents join), we could easily allow drivers to agree to several tasks in parallel, or let peers remove their previous requests if they are no longer interested in them.

## 5 Experimental Results

To establish whether the scalability and robustness we expect can actually be observed in a real-world implementation, we have developed a prototypical web-based system that runs the protocol depicted in figure 1, and evaluated it experimentally in the ridesharing domain. Our experiments below focus on the matchmaking and negotiation part of the protocol (from REQUEST to START), as this involves most dependencies among individual behaviours, and requires involves solving a complex combinatorial problem for the orchestrator agent $o$ that involves calculating exponential numbers of possible rides presented to every driver and passenger. Instead of trying to get agreement or rejection to a single potential ride from every peer involved, our architecture enables us to constantly update all rides available to every peer in the system. We also use two

further "non-standard" protocol operations: One is to automatically generate `INFORM_INVALID` messages for other participants when an agent `REJECTS` a ride, and the other is to delete all `INFORM_TASK` messages linked to a peer's request once a different ride for that peer has been agreed. Since in practice there is no global clock for synchronisation, all agents periodically poll the stores they are interested in (`INFORM_TASK` to check what the currently available rides are, and `START`/`INFORM_INVALID` to determine whether a ride has been agreed/can no longer be agreed). In terms of the execution engine, our implementation involves a single server which contains all message stores, and exposes operations on them through a simple RESTful Web API. The server runs `Node.js`, a non-blocking event-driven JavaScript library, and has separate processing queues associated with different message stores, which asynchronously process individual "platform jobs" for different client calls. Note that running the platform on a single server is not a requirement – in principle every message store could be located on a different server, including agent nodes that implement an HTTP interface.

Our first experiment examines the overall scalability of the platform. We create artificial "groups" of size $k$ in a population of $n$ agents such that all the requests inside a group match, and we can artificially control how many rides will be created. Our first experiment involves up to 10 groups of 6, 9, and 12 agents, i.e. a total of 60, 90, 120 agents, where the ratio of drivers $d$ to passengers $p$ is 1/2 (i.e. $p/d \in \{2/4, 3/6, 4/8\}$ for each group size). Note that the respective number of possible rides generated in each group is $(2^p - 1) * d$ as there is a different proposal for every subset of passengers, and the rides different drivers may offer to a group overlap. This means that 30/189/1020 rides have to be created for each group, i.e. the system has to deal with up to 10200 rides overall as we keep adding groups. Note also that, since all ride requests and agreements to rides occur in very close succession, the load of this system is similar to a real-world system that would experience this level of usage every few minutes (in reality, of course, users take much much longer to check updates and respond), so it is in fact representative of a very large scale real-world application. Finally, to maximise the amount of messages exchanged and the duration of negotiation, drivers accept only the maximally sized ride, and passengers accept all rides. The top two plots in figure 3 show the average time taken in seconds (across all agents, and for 20 repetitions for each experiment, with error bars to indicate standard deviations) for matchmaking (`REQUEST` and `INFORM_TASK`) and negotiation (all further messages up to and including `START`), respectively. As can be seen from these plots, even though every agent has a built-in delay of 2 seconds between any two steps, even when when there are 120 agents in the system, the average time it takes an agent to get information about all rides acceptable to her/complete the negotiation of a ride is around 50s/80s even in the largest configurations.

In the second experiment, we investigate the cumulative effect of adding delays and message failures on the total execution time of an entire negotiation for a ride, in order to assess how robust the system is. For this, we artificially increase the delay between any update an agent receives and its successive operation from 2s to 5s, 10s, and 20s. We use these artificial delays also to emulate failure, e.g. when network resources are temporarily unavailable. The bottom plot in figure 3 shows the results for this experiment, for a group size of 9 and 5 groups (45 agents in total), showing measurements for matchmaking, negotiation, and the total lifespan of an agent (from creation to agreement). As can be seen, the overall lifespan of an agent increases by a factor of 3 to 4 here when

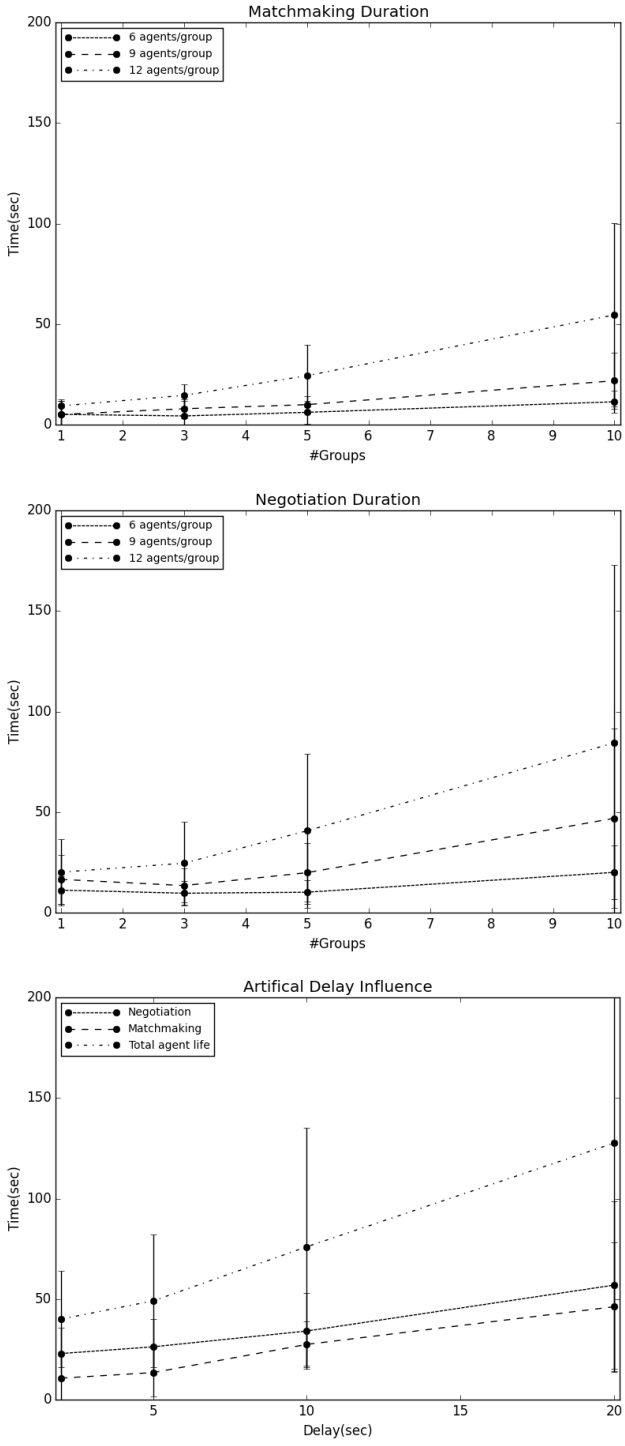



**Fig. 3.** Experimental results

the delay increases by a factor of 10, which is a good indication that the system degrades gracefully under increasing perturbation. Moreover, what is interesting is that the time taken for negotiation, which involves the highest number of messages to the orchestrator (as all passengers accept all rides) only increases by a factor between 1.5 and 2. This is because the larger delays require less effort for matchmaking and computing rides, and the orchestrator has more time to process negotiation-related messages during these gaps. This nicely illustrates how separating the processing of different message stores leads to effective load balancing for any agent that has to engage in different interactions concurrently.

## 6 Related Work

The idea of coordinating distributed processes through a shared coordination medium is not new. It can be traced back at least to the blackboard systems [5] used in early distributed knowledge-based systems. In distributed computing, similar ideas led to coordination languages like LINDA [8]. While these systems initially involved either fixed sets of coordination primitives or built-in, application-dependent coordination strategies, they were later used in platforms like TuCSoN [12] to develop programmable behaviours for the coordination data structures. Our approach differs from this line of work in that we do not attempt to replace the protocol-based interaction models used in mainstream agents research. Instead, we maintain their advantages in terms of supporting complex specifications of sequential interactions and agent communication language semantics. Mapping these onto a data-driven architecture gives us the "best of both worlds", as it allows us to capture complex agent interactions while separating coordination from computation.

An architecture that takes a similar protocol-centric approach to the regulation of agent behaviours is OpenKnowledge [13], which allows declarative specifications of interaction protocols to be directly executed in an open, peer-to-peer platform. While its automation of executing protocols from their specification is more advanced here than in our approach, it involves agents effectively handing over control to coordinators that "run" the agent processes (the agent can still make autonomous decisions regarding different choices available in the protocol, but the platform executes the protocol by invoking these local decision methods from outside). The difference to our approach is that we do not provide an execution platform that includes the agent processes themselves, but prefer to restrict the computational coordination process only to what is absolutely necessary.

Previous work most closely related to ours, however, is at the intersection of agents and service-oriented computing research. The authors of [2] present a method for mapping complex agent conversations to web services, without providing, however, a formal framework or a concrete implementation. Only very recently Singh [15, 16] addressed the service-based view of agent protocols by proposing a formal language and a computational architecture that supports it. His approach bears close resemblance to our work: It considers protocols in terms of information schemas without any additional control flow constructs, defines semantics in terms of histories of past message exchanges, and proposes an architecture that enables agents to asynchronously and atomically process individual messages, supporting distributed interactions that have multiple loci of enactment. The main difference to our approach is that the semantics provided in this model does not take account of non-message actions and local state transitions, instead focusing more on protocol verification. Also, no quantitative performance results for an implementation of this system are presented.

It is worth mentioning, that, with the exception of [15], all of the above approaches involve some kind of *middleware* that relies on a bespoke communication architecture and platform that the agents must comply with. Moreover, while these platforms could be exposed, at least in principle, over normal Web APIs, agent designers would still have to be familiar with the specific languages used by them. In our framework, we do not only do away with such specific middleware. We also reduce the language specification for messages and constraints to a very general form, through constraints that are general variable restrictions, and messages with simple pre- and postconditions. As long as ontological agreement can be assumed regarding the semantics of individual variables and their domains (which is also a prerequisite for all of the above approaches), our APIs should be straightforward to use. In these respects, our work is heavily influenced by the REST paradigm [6], in that it uses ordinary Web resources as the means of exposing state to peers in order to coordinate the workflow between them. To our knowledge, there have been no attempts to formalise the semantics of this paradigm, and while our work does not aim to provide such semantics for the general case, it can be seen as a contribution toward a better overall understanding of REST itself.

## 7 Conclusions

In this paper, we have presented a data-driven architecture for coordinating agent collectives on the Web that is aimed at bridging the gap between work on agent interaction protocols and modern Web-based applications used commonly in areas such as social and collective computation. We presented a formal framework that allows us to specify the semantics of our architecture, and which allowed us to introduce functionality that is not available in normal agent-based systems platforms. Our experimental results with a prototypical implementation show that it can handle complex interactions in a lightweight way, producing minimal overhead while providing good scalability and robustness properties.

We summarise the main benefits of our approach: Firstly, there are no sequential distributed processes that need to rely on a standing line of communication, since all data operations are atomic, and can be easily repeated in case of failure. As our experiments show, the overhead of the additional link structure that has to be stored and the frequent "pull" operations from agents do not seem to affect performance significantly. Secondly, in a real Web deployment, we could directly benefit from the standard caching facilities of Web servers that can store frequently reused resources. Thirdly, coordination platforms can cross-check parallel interactions and apply global constraints to the overall interactions flexibly. Fourthly, since all operations are atomic, the decision logic can be devolved to components processing data in parallel whenever different steps are independent. This also provides guidance for designing agents' internal reasoning mechanisms, or for "splitting" functionality into several agents. Finally, message stores and the linkage between them provide a direct "data" view to the ongoing interactions at a global level, thus facilitating analysis, prediction, and ease of mapping to other structures such as provenance information. In fact, all operations in our architecture can easily be captured using standard formats like `PROV`[6], and our implementation supports this through full integration with a live `PROV` server.

As regards future work, on the practical side, we plan to focus on developing an automated procedure for generating implementations of our architecture

[6] See `http://www.w3.org/TR/prov-overview/`

directly from a given protocol specification. On the more theoretical side, we would like to develop formal procedures to detect and decouple different parts of a coordination protocol where these are not causally linked.

## 8 Acknowledgments

The research presented in this paper has been funded by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 600854 "*SmartSociety – Hybrid and Diversity-Aware Collective Adaptive Systems: Where people meet machines to build smarter societies*" (http://www.smart-society-project.eu/).

## References

1. Ahmad, S., Battle, A., Malkani, Z., Kamvar, S.D.: The Jabberwocky Programming Environment for Structured Social Computing. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology. pp. 53–64. Santa Barbara, CA (2011)
2. Ardissono, L., Goy, A., Petrone, G.: Enabling conversations with web services. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003). pp. 819–826 (2003)
3. Chopra, A.K., Artikis, A., Bentahar, J., Colombetti, M., Dignum, F., Fornara, N., Jones, A.J.I., Singh, M.P., Yolum, P.: Research Directions in Agent Communication. ACM Transactions on Intelligent Systems and Technology 4(2), 1–23 (2013)
4. Crosby, M., Rovatsos, M., Petrick, R.: Automated Agent Decomposition for Classical Planning. In: Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013). pp. 46–54. Rome, Italy, June 10-14 (2013)
5. Englemore, R., Morgan, T. (eds.): Blackboard Systems. Addison-Wesley, Reading, MA (1988)
6. Fielding, R.T., Taylor, R.N.: Principled Design of the Modern Web Architecture. ACM Transactions on Internet Technology 2(2), 115–150 (2002)
7. Fikes, R., Nilsson, N.: STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2(3-4), 189–208 (1971)
8. Gelernter, D., Carriero, N.: Coordination languages and their significance. Communications of the ACM 35(2), 97–107 (1992)
9. Law, E., von Ahn, L.: Human Computation. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers (2011)
10. Little, G., Chilton, L.B., Goldman, M., Miller, R.C.: TurKit: Human Computation Algorithms on Mechanical Turk. In: Proceedings of the 23nd Annual ACM Symposium on User Interface Software and Technology. pp. 57–66 (2010)
11. Malone, T.W., Laubacher, R., Dellarocas, C.: The collective intelligence genome. Sloan Management Review 51(3), 21–31 (2010)
12. Omicini, A., Zambonelli, F.: Tuple centres for the coordination of internet agents. In: Proceedings of the 4th ACM Symposium on Applied Computing. San Antonio, TX (February 1999)
13. Robertson, D., Barker, A., Besana, P., Bundy, A., Chen-Burger, et al: Sierra, C., Walton, C.: Models of interaction as a grounding for peer to peer knowledge sharing. In: Advances in Web Semantics I. Lecture Notes in Computer Science, vol. 4891. Springer-Verlag, Berlin, Heidelberg (2007)
14. Rovatsos, M.: Multiagent systems for social computation. In:Proceedings of the 13th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014). Paris, France (2014)
15. Singh, M.P.: LoST: Local State Transfer - An Architectural Style for the Distributed Enactment of Business Protocols. In: Proceedings of the 9th International Conference on Web Services (ICWS). pp. 57–64. Washington, DC (2011)
16. Singh, M.P.: Semantics and Verification of Information-Based Protocols. In: Proceedings of the Eleventh International Conference on Autonomous Agents Multiagent Systems (AAMAS 2012). pp. 1149–1156. Valencia, Spain, June 4-8 (2012)