Università degli Studi di Bologna
DEIS

Compliance Verification of Agent Interaction: a Logic-based Software Tool

Marco Alberti Federico Chesani
Marco Gavanelli Evelina Lamma
Paola Mello Paolo Torroni

September 20, 2004

DEIS Technical Report no. DEIS-LIA-004-03 *LIA Series no. 71*

Compliance Verification of Agent Interaction: a Logic-based Software Tool

Marco Alberti [1] Federico Chesani [2] Marco Gavanelli [1]
Evelina Lamma [2] Paola Mello [1] Paolo Torroni [1]

[1] *Dipartimento di Ingegneria, Università di Ferrara*
Via Saragat, 1
44100 Ferrara, Italy
{malberti, mgavanelli, elamma}@ing.unife.it
[2] *DEIS, Università di Bologna*
V.le Risorgimento 2
40136 Bologna, Italy
{fchesani, pmello, ptorroni}@deis.unibo.it

September 20, 2004

Abstract. In open societies of agents, where agents are autonomous and heterogeneous, it is not realistic to assume that agents will always act so as to comply to interaction protocols. Thus, the need arises for a formalism to specify constraints on agent interaction, and for a tool able to observe and check for agent compliance to interaction protocols. In this paper we present a Java-Prolog software component built on logic programming technology, which can be used to verify compliance of agent interaction to protocols, and that has been integrated with PROSOCS [Stathis *et al.*, 2004].*

Keywords: *computational logic, logic programming, multi-agent communication, agent interaction, protocol verification, implementation*

1 Introduction

Agent interaction in multiagent systems is usually ruled by interaction protocols. In open societies of agents, where agents can be heterogeneous and, in general, their

*This article will appear in a Special Issue of *Applied Artificial Intelligence*, Taylor & Francis (2005)

internals cannot be accessed, it is not realistic to assume that agents are built so as to always be compliant to interaction protocols. In this perspective, the need arises for a formalism to constrain the agents' *observable* behaviour rather than their internal (mental) structure or state, and for a tool to verify compliance of agent interaction to a given specification.

The social approach to the definition of interaction protocols and semantics of Agent Communication Languages is a noteworthy attempt to meet these requirements. Significant contributions have been given by Yolum and Singh [2002], Artikis et al. [2002], Fornara and Colombetti [2003] and Verdicchio and Colombetti [2003].

In previous work [Alberti *et al.*, 2003c], we proposed to this purpose a logic-based formalism, called *Social Integrity Constraints* (ic_S). ic_S can be used to provide semantics to communicative actions and protocols which define the agent interaction in an open social environment. Such a semantics is given in terms of expectations about the behaviour of agents based on a history of observed actions. ic_S can be viewed as integrity constraints in an abductive framework [Alberti *et al.*, 2003b], so as to exploit well-established results from Abductive Logic Programming [Kakas *et al.*, 1993], and define a correct proof-procedure that can be used for verification of compliance of agent behaviour to a specification. The specification can, thus, also be interpreted as an abductive logic program for verification of compliance. This approach is described in [Alberti *et al.*, 2003d].

In this paper, we describe *SOCS-SI*[1]: a Java-Prolog-CHR based implementation of the interaction verification framework based on the proof-procedure defined in [Alberti *et al.*, 2003d]. The intended use of *SOCS-SI* is in combination with agent platforms, such as PROSOCS [Stathis *et al.*, 2004], in a way that allows for on-the-fly verification of compliance to protocols. In *SOCS-SI*, the proof-procedure is part of an integrated environment, which also contains interface modules to allow for such a combination, and a Graphical User Interface (GUI). The GUI provides an intuitive way to observe the actual behaviour of the society members with respect to their expected behaviour, and to detect possible deviations.

Following the description of *SOCS-SI*, we show through examples its possible use in the design of the agent interaction space, in terms of Agent Communication Language semantics and Interaction Protocols and properties definition.

The paper is structured as follows. In Sect. 2, we give a brief, informal introduction to the framework and to the proof-procedure. In Sect. 3, we present the implementation of the proof-procedure. In Sect. 4 we demonstrate the use of SOCS-SI for interaction space design in open agent systems. Discussion of related work and conclusions follow.

[1] "*SOCS*" is the acronym of the EU-funded project (IST-2001-32530) that partially supported this work [SOCS, 2001]. *SI* stands for *Social Infrastructure*.

2 Logic-based Specification and Verification

In this section we give motivations and the necessary background on the formal framework proposed by Alberti et al. [2003a; 2003c; 2003b] for the specification of agent interaction in open societies of agents.

Various definitions have been given in the literature for *openness*. According to Davidsson [2001], in an open (artificial) society "there are no restrictions for agents/processes to join/leave the society". According to Artikis et al. [2002] a society is open if it does not constrain the agents to be based on a same technology or framework, and can possibly be non-cooperative. In both cases, we cannot assume, for example, that in an open society all agents have *beliefs, desires* and *intentions* [Kinny *et al.*, 1996], and, even if they do, we cannot assume to have access to their internals without severely undermining their autonomy. Thus, the check of compliance of agents to protocols can be performed only by observing their externally visible behaviour.

The framework assumes the existence of an entity (*Social Compliance Verifier* or SCV, for short) which is external to agents, and is devoted to check their compliance to the specification of agent interaction.

The SCV is aware of the ongoing social agent interaction: this is represented by a set of (ground) facts called *events*, and indicated by functor **H**. For example, $\mathbf{H}(request(a_i, a_j, give(10€), d_1), 7am)$ represents the fact that agent a_i requested agent a_j to give 10€, in the context of interaction d_1 (dialogue identifier) at time $7am$.[2]

In open agent societies, the agent behaviour is unpredictable, because agents are autonomous. However, interaction protocols can be defined to let autonomous agents interact in a social context. In that case, it becomes possible to reason upon the expected (future) social behaviour of agents, given the observations made on their current behaviour and the defined social protocols. If protocols are engineered so as to define the "desirable" patterns of social interaction, e.g. by having in mind the properties that a system of agents complying to such protocols will exhibit, expectations represent in some sense the "ideal" behaviour of a society. Expectations can be positive (events expected to happen, indicated by the functor **E**) or negative (events expected *not* to happen, functor **EN**). Expectations have the same format as events, but they will, typically, contain variables, to indicate that expected events are not completely specified, leaving freedom degrees to the agents. Constraints à la CLP (Constraint Logic Programming [Jaffar and Maher, 1994]) can be imposed on variables, in order to refine and restrict the focus of the expectations. For instance,

$$\mathbf{E}(accept(a_k, a_j, give(M), d_2), T_{\mathrm{a}}), M \geq 10€, T_{\mathrm{a}} \leq 11pm$$

[2] We make the simplifying assumption about time of events, that the time of sending a message is the same as receiving it, and that such time is assigned by the social framework.

represents the expectation for agent a_k to *accept* giving agent a_j an amount M of money, in the context of interaction d_2 (dialogue identifier) at time T_a; CLP constraints say that M is expected to be greater or equal than 10€, and T_a to be not later than $11pm$.

Given a set of observed events and the current expectations, *Social Integrity Constraints* (ic_S) define what new expectations are to be generated in the society.

Let us consider an example with two agents involved (although ic_S can be applied to any-party agent interaction):

$$\begin{aligned} &\mathbf{H}(\mathit{request}(A,B,P,D),T_1) \\ \rightarrow&\mathbf{E}(\mathit{accept}(B,A,P,D),T_2), T_2 \leq T_1 + \tau \\ \vee&\mathbf{E}(\mathit{refuse}(B,A,P,D),T_2), T_2 \leq T_1 + \tau \end{aligned} \tag{1}$$

states that, if agent A *request*s P to agent B, in the context of interaction D at time T_1, then agent B is expected to *accept* or *refuse* P by τ time units after the *request*.

The following ic_S:

$$\begin{aligned} &\mathbf{H}(\mathit{accept}(A,B,P,D),T_1) \\ \rightarrow&\mathbf{EN}(\mathit{refuse}(A,B,P,D),T_2), T_2 \geq T_1 \end{aligned} \tag{2}$$

$$\begin{aligned} &\mathbf{H}(\mathit{refuse}(A,B,P,D),T_1) \\ \rightarrow&\mathbf{EN}(\mathit{accept}(A,B,P,D),T_2), T_2 \geq T_1 \end{aligned} \tag{3}$$

express, instead, mutual exclusiveness between *accept* and *refuse*: if an agent performs an *accept*, it is expected *not* to perform a *refuse* with the same content after the *accept*, and vice versa. In this way, we are able to define protocols as sets of forward rules, relating events to expectations.

In this perspective, observed social events represent known facts about the social behaviour of agents, whereas expectations represent hypotheses about their ideal future behaviour. The formal machinery that we chose to adopt in this context is therefore one supporting hypothetical reasoning.

Abduction [Kakas *et al.*, 1993] is a reasoning paradigm which consists of formulating hypotheses (called *abducibles*) to account for observations; in most abductive frameworks, *integrity constraints* are imposed over possible hypotheses in order to prevent inconsistent explanations. The idea behind our framework is to formalise expectations about agent behaviour as abducibles, and to use Social Integrity Constraints such as (1), (2) or (3) to rule out, among the possible expected agent behaviours, those violating the interaction protocols. Thanks to the abductive framework, goal-directed behaviour of the society can be easily implemented. For example, the society might request that at least one exchange should be executed successfully before $7pm$, and have the following goal:

$$\mathbf{E}(\mathit{accept}(B,A,P,D),T), T \leq 7pm.$$

Given the partial history of a society, an abductive proof-procedure, $\mathcal{S}$CIFF [Alberti *et al.*, 2003d], generates expectations about agent behaviour so as to comply with Social Integrity Constraints. $\mathcal{S}$CIFF is inspired by the IFF proof-procedure [Fung and Kowalski, 1997], augmented as needed to manage CLP constraints. The most distinctive features of $\mathcal{S}$CIFF are, however, (*i*) its ability to check that the generated expectations are *fulfilled* by the actual agent behaviour (i.e., that events expected (not) to happen have actually (not) happened), which cannot be assumed *a priori* in an open society of autonomous agents, and (*ii*) its ability to raise exceptions, called *violations* when the expectations are not met.

3 The *SOCS-SI* Tool

The implementation of the *SOCS-SI* tool for compliance verification of agent interaction is composed of an implementation of the proof-procedure specified in [Alberti *et al.*, 2003d], interfaced to a graphical user interface and to a component for the observation of agent interaction.

The *SOCS-SI* software application is composed of a set of modules. All the components except the proof-procedure are implemented in the Java language.

The core of *SOCS-SI* is composed of three main modules (see Fig. 1), namely:

- *Event Recorder*: fetches events from different sources and stores them inside the *History Manager*.
- *History Manager*: receives events from the *Event Recorder* and composes them into an "event history".
- *Social Compliance Verifier*: fetches events from the *History Manager* and passes them on to the proof-procedure in order to check the compliance of the history to the specification.

In our model, agents communicate by exchanging messages, which are then translated into **H** events (see Sect. 3.3). The *Event Recorder* fetches events and records them into the *History Manager*, where they become available to the proof-procedure (see Sect. 3.1). As soon as the proof-procedure is ready to process a new event, it fetches one from the *History Manager*. The event is processed and the results of the computation are returned to the GUI. The proof-procedure then continues its computation by fetching another event if there is any available, otherwise it suspends, waiting for new events.

A fourth module, named *Init&Control Module* provides for initialisation of all the components in the proper order. It receives as initial input a set of protocols defined by the user, which will be used by the proof-procedure in order to check the compliance of agents to the specification.

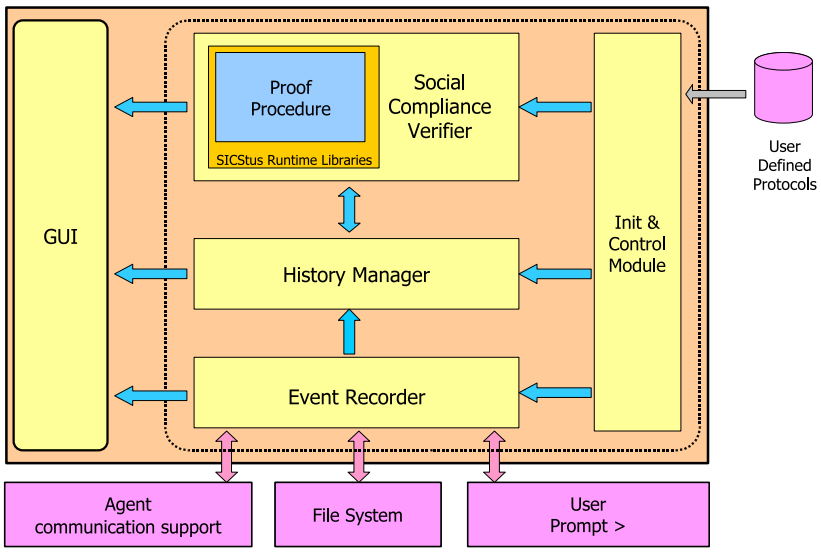

Figure 1. *Overview of the SOCS-SI architecture*

3.1 Implementation of the proof-procedure

As its ancestor, the IFF proof procedure [Fung and Kowalski, 1997], the $\mathcal{S}$CIFF proof-procedure is a transition system that rewrites logic formulae into equivalent logic formulae. Each formula is a *Node* of the proof-procedure, and can be rewritten into one or more nodes, logically in OR (so building an OR-tree). Elements in a formula (node) are arranged in a tuple that carries the following information:

$$T \equiv \langle R, CS, PSIC, \mathbf{EXP}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle \tag{4}$$

where R is the resolvent, CS is the constraint store (as in CLP), PSIC is a set of implications (initially set as $\mathcal{IC}_S$), **HAP** is the current history, **EXP**, **FULF**, and **VIOL** are, respectively, the set of pending, fulfilled, and violated expectations.

For the implementation of the $\mathcal{S}$CIFF proof-procedure, SICStus Prolog [SICStus, 2003] has been chosen, for the following reasons:

- the Prolog language offers built-in facilities for the implementation of dynamic data structures and (customisable) search strategies;
- SICStus Prolog allows for state-of-the-art CLP; in particular, the libraries CHR, CLPB, and CLPFD have been exploited;
- SICStus Prolog features a bidirectional Java-Prolog interface (Jasper), which has been necessary to interface the proof-procedure with the other modules of the social demonstrator.

As the IFF proof-procedure [Fung and Kowalski, 1997], the $\mathcal{S}$CIFF proof-procedure [Alberti *et al.*, 2003d] specifies the proof tree, leaving the search strategy to be defined at implementation level. The implementation is based on a depth-first strategy. This choice, enabling us to tailor the implementation for the built-in computational features of Prolog, allows for a simple and efficient implementation of the proof-procedure.

The Prolog-CHR module implements the transitions of the proof-procedure. CHR [Frühwirth, 1998] is a rewriting system for implementing new constraint solvers. It is based on forward rules that rewrite constraints into other constraints. For example, in order to define the solver for the constraint $\leq$, one can write the following rules:

$$\begin{array}{rcrcl} \mathit{antisymmetry} & @ & A \leq B, B \leq A & <=> & A = B \\ \mathit{transitivity} & @ & A \leq B, B \leq C & ==> & A \leq C \end{array}$$

The first rule is a *simplification* rule, and says that if both $A \leq B$ and $B \leq A$ are in the constraint store, then you can *replace* both of them with the new constraint $A = B$ (in CLP, unification is a constraint). The second is a *propagation* rule, *adds* a new constraint $A \leq C$ when both $A \leq B$ and $B \leq C$ are in the store. Thanks to these two simple rules, the constraints $X \leq Y$, $Y \leq Z$, $Z \leq X$ are resolved and $X = Y = Z$ is inferred.

The data structures of the proof-procedure (eg. PSIC, **EXP**) are implemented as CHR constraints, so the transitions can be straightforwardly implemented as CHR rules. For example, each happened event is represented by means of a `h/2` CHR constraint, whose (ground) arguments are the content and the time of the event. An example of event is:

```
h(request(seller,buyer,give(10€),1),10am)
```

Expectations are represented by means of CHR constraints `e` for **E** expectations and `en` for **EN** expectations. CHR interfaces easily with other constraint solvers, so we can impose constraints on the variables, like

```
e(do(buyer,seller,give(10€),1),T), T<5pm
```

and this expectation will not match with (and be fulfilled by), for example, a happened event

```
h(do(buyer,seller,give(10€),1),8pm).
```

Given this representation, the $\mathcal{S}$CIFF transitions can be mapped into CHR rules, in a sense defining a new constraint solver for the resolution of expectations. For example, we have a transition of **E**-consistency, that ensures that the final derivation node does not contain both the expectations of an event to happen and to not happen:

```
e_consistency @
        e(E1,T1), en(E2,T2)
        ==>
        reif_unify((E1,T1),(E2,T2),0).
```

Such a rule, for each pair $(\mathbf{E}(E_1, T_1), \mathbf{EN}(E_2, T_2))$ imposes the dis-unification constraint $(E_1, T_1) \neq (E_2, T_2)$.

The fulfillment rule is also rather straightforward:

```
fulfillment @
    h(HEvent,HTime), e(EEvent,ETime)
    ==>
    may_unify(HEvent,EEvent)
    |
    renaming((EEvent,ETime),(EEvent1,ETime1)),
    case_analysis_fulfillment((HEvent,HTime),(EEvent,ETime)).
```

The rule is applied when an event and a pending expectation whose content have the same functor and arity (checked by the `may_unify/2` predicate in the guard of the rule) are in the CHR store. In this case, a renaming is made of the expectation[3] and the `case_analysis_fulfillment/7` predicate is called. Two nodes are created by `case_analysis_fulfillment/7`:

- a first node where unification is imposed between the expectation and the event, the `e(EEvent,ETime)` constraint for the expectation is removed from the constraint store and the `fulf(e(EEvent,ETime))` CHR constraint is imposed (implementing the fact that the expectation is moved from the set **EXP** of pending expectations to the **FULF** one of fulfilled expectations);
- and a second node where dis-unification between the expectation and the event is imposed.

3.2 The Java-Prolog Interface

The main task of the Java portion of the *Social Compliance Verifier* is to interact with the proof-procedure. The SICStus Runtime libraries are accessed from Java using the Jasper package and native interfaces. All data exchanged between the Java sides and the Prolog program is translated into String objects. In order to process and filter the String objects, Java regular expressions are extensively used. These expressions are defined in a configuration file, loaded at initialisation time.

[3] This step is necessary because some expectations may contain universally quantified variables. The issue is discussed in detail in a technical report [Alberti *et al.*, 2003d].

Our software application can deal with different proof-procedure implementations and with different ACL performatives, without any a priori assumption about the format of the exchanged parameters. It is sufficient to properly re-define the regular expressions in the config file, and a new proof-procedure can be easily integrated into the software application.

3.3 Messages vs. Events

While the proof-procedure can deal with events of any format that can be represented as a Prolog term, for the purposes of this work we can assume that the agents communicate by exchanging "messages", where a message is defined by the following data set:

- a sender
- a receiver (one or more than one)
- a dialogue identifier
- a time
- a communication performative
- a list of parameters of such a performative

Our software can deal with any platform for agents, as long as the communication between agents can be represented in such a way. Inside the application, each message is translated into an "event".

3.4 The Recorder Interface

The *Event Recorder* fetches events from the external world using modules, each module being specialised for a specific source. We developed modules for interfacing with agent platforms, starting with PROSOCS [Stathis *et al.*, 2004]. We are currently experimenting with other platforms: we had some successful experiments with JADE [Bellifemine *et al.*, 2000] and TuCSoN [Ricci *et al.*, 2002], and with checking compliance of e-mail messages. For testing and debugging purposes, we also developed modules to interact with the user prompt, as well as with the file system; it is possible to add as many specialised modules as desired, provided that they implement the interface `RecorderInterface`. In order to integrate our application with an already existing platform the user should:

1. create a Java class that implements the `RecorderInterface`
2. select it as message source during the application configuration (either through the configuration GUI, or by modifying the config file).

The `RecorderInterface` that we propose defines three methods, where the class `SOCSEvent` is our internal representation of events:

- `public SOCSEvent listen()`. Returns an instance of the `SOCSEvent` class if a message is available, or it waits (suspends) until a message arrives.
- `public long speak(SOCSEvent aMsg)`. Gives our application the capability to communicate with agents, by sending a message. It returns the time the message is sent.
- `public long getTime()`. Returns the current time. It is used to check temporal deadlines.

The `RecorderInterface` has originally been defined as a subset of the low level communication API defined in the PROSOCS platform [Stathis *et al.*, 2004], which is used to perform controlled experiments in the context of global computing applications, within the SOCS project [SOCS, 2001]. However, one of the design specifications we strove to obtain was to have an interface general enough to allow integration with most agents platforms currently available.

3.5 The Graphical User Interface

The Graphical User Interface is implemented by using the Swing graphic library, and implements the Model-View-Control programming pattern. The main window is composed of three areas (or sub-windows), and of a button bar containing the controls (Figure 2).

The bottom area contains the list of all the messages received by the *SOCS-SI*: the next message to be processed by the proof-procedure is emphasised (in Fig. 2 it is the third row, which is darker). The area on the left contains the list of agents known by the society, i.e., agents that have performed at least one communicative action (coherently with the notion of openness by Davidsson [2001]). The larger frame on the right contains the results of the computation, returned by the proof-procedure. These results are expressed in terms of society expectations about the future behaviour of agents, and also in terms of fulfilled expectations and violations of social rules. By selecting an agent from the left pane, it is possible to restrict the information shown on the larger pane to be only that relevant to that particular agent. Among other features, it is possible to execute step-by-step the application, so that it elaborates one message at a time and then waits for a user acknowledge (similarly to the debug interface of modern compilers).

Protocols are loaded into the tool by means of a button; they are simply provided as text files with a syntax strictly adherent to the formal one presented earlier. For example, Equation 1 is rewritten (in the case $\tau = 3$) as:

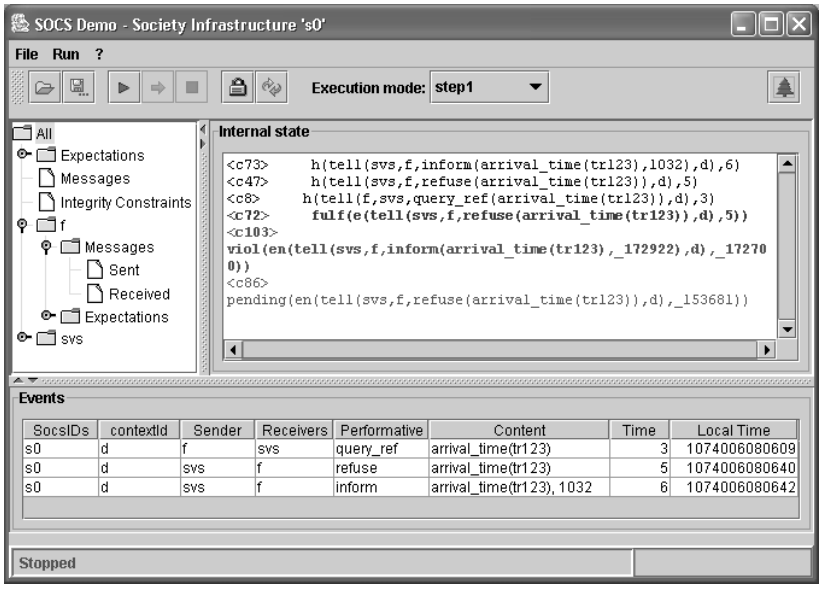

Figure 2. *A screenshot of the application*

```
 H(tell(A, B, request(P), D), T1)
--->
E(tell(B, A, accept(P), D), T2) /\ T2 <= T1 + 3
\/
E(tell(B, A, refuse(P), D), T2) /\ T2 <= T1 + 3.
```

Finally, a tree-view of the whole computation is provided (Figure 3); interestingly, the shown tree bears both an operational and a logical interpretation.

The operational interpretation is an intuitive graphical form of a log-file, showing the most significant computational steps, useful for debugging purposes.

The logical meaning is an or-tree of the possible derivations timed by the incoming events. For each incoming event that enriches the knowledge base, the frontier of the explored proof-tree (which is a logical disjunction, as in various proof-procedures [Fung and Kowalski, 1997]) is shown.

The user can inspect each of the nodes, and see in the main window the state of the computation (i.e., the above given tuple, Eq. 4), having the logical interpretation of a conjunction (of logical formulae of the types in the $\mathcal{S}$CIFF language: abducibles, constraints, literals, implications).

Presentation of the frontier of the derivation tree is important for explanation reasons. Typically, logic languages can provide two types of answers: a *success/failure*

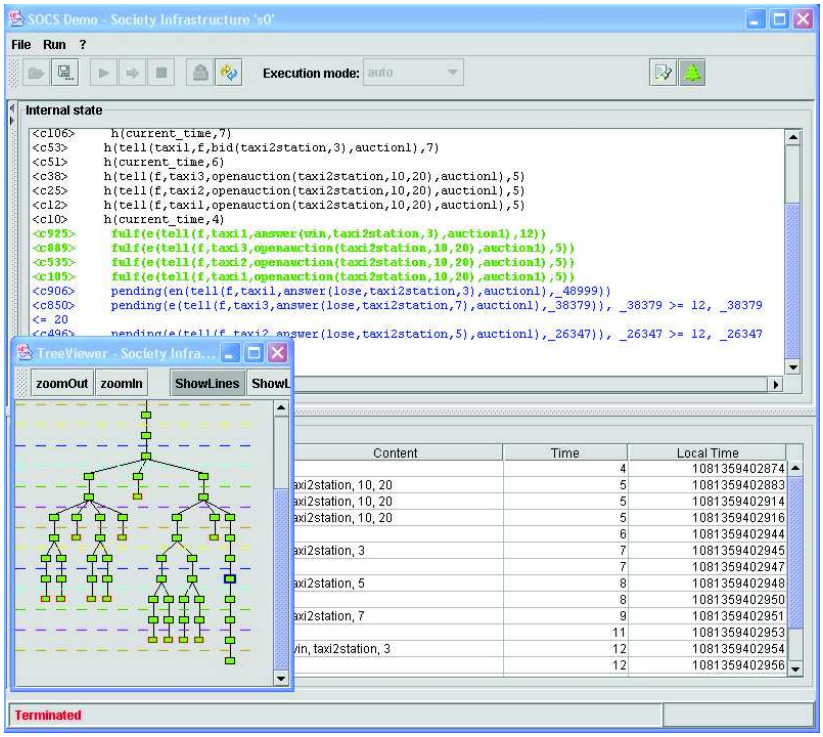

Figure 3. *A screenshot of the application*

answer and an *explanation* answer. In case of success, logic languages explain *why*: Prolog returns simply a binding for the variables in the goal, CLP can return also constraints, and ALP (Abductive Logic Programming) returns a set of abducibles, just to name a few. But in case of failure, there is typically no explanation.

The tree-view provides information also in case of failure: the set of failing nodes, each with underlined the cause of failure (e.g., a violated expectation or an unsatisfiable CLP constraint).

Example 3.1 *As an example, let us consider the* request-accept-refuse *protocol (Eq. 1) with deadline* $\tau = 3$*, and let us assume that the following events have happened:*

$$\mathbf{HAP} = \{\mathbf{H}(request(a, b, give(10€), d_1), 1pm), \tag{5}$$

$$\mathbf{H}(refuse(b, a, give(10€), d_1), 10pm)\}. \tag{6}$$

As the first event happens (at 1pm), the $\mathcal{SCIFF}$ *proof-procedure generates two branches (Figure 3.1): in one it assumes that agent b will reply* accept*:*

$$\mathbf{E}(accept(b, a, give(10€), 1), T), T < 4pm \tag{7}$$

$\emptyset$			
$\mathbf{H}(request, 1)$	$\mathbf{E}(accept, T)$ $T < 4$	$\mathbf{E}(refuse, T)$ $T < 4$	
$\mathbf{H}(refuse, 10)$	$\underline{\mathbf{E}(accept, T)}$ $T < 4$	$\mathbf{E}(refuse, T)$ $\underline{T = 10 \wedge T < 4}$	$\underline{\mathbf{E}(refuse, T)}$ $T \neq 10 \wedge T < 4$

Figure 4. *Example of proof-tree*

in the second assumes that b replies refuse *within the given deadline*

$$\mathbf{E}(refuse(b, a, give(10€), 1), T), T < 4pm. \tag{8}$$

When the event (6) happens, the proof-procedure tries to match it with the raised expectation in each of the branches. In the first branch, accept$(\dots)$ *cannot unify with refuse*$(\dots)$*, so the expectation (7) is declared* violated *and is the culprit of failure in this branch.*

In the second branch, the proof-procedure generates two children:

- *in one it will assume that the raised expectation (8) matches with the event (6); in this case it will try the unification* $T = 10pm$*, that does not satisfy the CLP constraint* $T < 4pm$*: the culprit of failure is, in this case, the inconsistent constraint store*
- *in the other, it assumes that (8) does not match with (6), and impose* $T \neq 10pm$*; but, in this case, there is no event fulfilling the expectation, so there is a failure and the culprit is the violated expectation (8).*

Now, besides the description of the operational behaviour of the $\mathcal{S}$*CIFF proof-procedure, the frontier of the tree logically explains* why *the computation failed:*

$$\begin{array}{lll} & violated & \mathbf{E}(accept(b, a, give(10€), 1), T) \\ \wedge & violated & \mathbf{E}(refuse(b, a, give(10€), 1), T), T \neq 10pm \\ \wedge & inconsistent & T = 10pm \wedge T < 4pm \end{array}$$

i.e., there is no such event matching with the first expectation, and there is no such event matching with the second, and it is inconsistent that T *is 10pm and, at the same time, before 4pm.*

The conjunction of the culprits in the alternative leafs can be thought as one explanation of the failure. Such an explanation is currently not minimal: in one node there could be more reasons for a failure, and one of them could be common for all the nodes in the frontier, so it would be the minimal explanation of failure. For efficiency motivations, only the first encountered reason is emphasised and taken as the culprit. We are currently working towards providing a minimal set of culprits.

4 Using *SOCS-SI* for interaction design in open multi-agent systems

In the previous sections we have presented *SOCS-SI* as a tool to verify the compliance of heterogenous agents to some defined social protocols, which are part of the definition of an existing social infrastructure. In this section, we show another possibility to use *SOCS-SI*, taking a different perspective. We will assume that the protocols are not yet defined, and consider the problem of designing the agent interaction space: a challenging aspect in the engineering of open systems.

The study and specification of interaction has since long been a central topic of agent research. In recent years, considerable effort has been devoted to developing on the one hand methodologies and tools for the verification of agent interaction, on the other hand languages and formalisms for the specification of Agent Communication Language (ACL) semantics and Interaction Protocols (IPs). Work on ACLs typically aims at providing the tools for heterogeneous agents to interact with each other by using a common set of communication "primitives", with a well understood semantics. On top of ACLs, IPs define the allowed/expected sequences of communication primitives that constitute an articulated interaction among a number of agents, be it a dialogue or a multi-party interaction, such an auction. Work on protocols is greatly devoted to defining mechanisms in such a way that some important properties are guaranteed. Some examples of properties are incentive-compatibility in the case of auction mechanisms, and termination in the case of negotiation dialogues. In the following, we will refer to the conjunct of ACLs and IPs as to the agent "interaction space".

As discussed by Omicini and Ossowski [2003], following Gelernter and Carrero [1992], the agent interaction space could be designed using a subjective perspective, derived from the agent specifications, or using an objective coordination model, independently of the agents which will populate the system. If we follow an objective perspective, differently from other agent oriented software methodologies where engineering societies mainly aims at implementing agents, we can focus on proving properties of agent interaction, independently of the very agents. The design of the interaction space can be then described as an iterated process consisting of the following phases as it has been described in [Torroni *et al.*, 2004]:

1. definition/refinement of the *environment* (agents systems and interaction media): this can be done following another agent-oriented software engineering methodology, such as those cited above;

2. definition/refinement of the *interaction space* (in particular, ACL semantics and protocol specification);

3. definition/refinement of formal *properties* that we would like the system to

exhibit, and their *verification*;

4. if properties are disproved, back to phase 1.

Once a model is done which satisfies the properties that we have defined, it can be implemented into a concrete agent system. We will now show the use of *SOCS-SI* within the above methodology, adopting as a running example the design of an auction interaction setting.

4.1 Environment definition

The *environment* is composed by the agents themselves, the communication media, and by the contextual entities that are relevant to the operation of the agent system.

In an auction example, the environment could include a variable number of agents having one of two possible roles (n bidders, $n \geq 1$, and one auctioneer), and a communication medium that permits bidirectional communication between auctioneer and each bidder.

SOCS-SI does not play a role in this step, which could well be done by following any approach from literature, such as the Gaia methodology [Wooldridge *et al.*, 2000], the KGR approach [Kinny *et al.*, 1996], or the Agentis approach [d'Inverno *et al.*, 1998], and possibly achieving a first definition of the entities agents, and a concrete realisation of the multi-agent system. When we proceed to the subsequent phases, we might find out that the current definition of the environment does not allow for modelling a system which exhibits the properties that we are interested in. In that case, this step should be iterated starting from different assumptions.

4.2 Interaction space definition

The *interaction space* is defined in terms of ACLs and IPs. This can be done by using *Social Integrity Constraints* as a uniform means to specify both ACL semantics and IPs.

In order to put things more concretely, let us give the specification of an auction context, where agents interact by proposing items on sale, bidding values to buy such items, and notifying the winner. The semantics of communicative acts such as *bid* and *win* can be defined as their intended social meaning.

Intuitively, the semantics of *"bid"* is a commitment to pay the declared amount of

money for an item. A possible way to define such a semantics could be the following:

$$\begin{aligned} &\mathbf{H}(tell(B, A, bid(Item, Q), D), T_{Bid}), \\ &\mathbf{H}(tell(A, B, answ(win, Item, Q), D), T_{Win}), \\ &\mathbf{H}(tell(A, B, deliver(Item), D), T_{Del}) \\ \rightarrow &\mathbf{E}(tell(B, A, pay(Item, Q), D), T_{Pay}), \\ &T_{Pay} < T_{Del} + T_{Pay\_Deadline} \end{aligned} \tag{9}$$

The semantics of the communication primitive "bid" formally defines in (9) what could be informally read as follows: "If an agent A makes a *bid* for some *Item*, and the auctioneer responds to such a bid by telling *win*, then if A is delivered the *Item*, he is expected to *pay* by some deadline". Similarly by answering *"win"*, the auctioneer declares his will to provide the item:

$$\begin{aligned} &\mathbf{H}(tell(B, A, bid(Item, Q), D), T_{Bid}), \\ &\mathbf{H}(tell(A, B, answ(win, Item, Q), D), T_{Win}), \\ \rightarrow &\mathbf{E}(tell(A, B, deliver(Item), D), T_{Del}), \\ &T_{Del} < T_{Win} + T_{Deliver\_Deadline} \end{aligned} \tag{10}$$

Various types of auction protocols are used in human and agent commerce; take, for example, the English auction, the Dutch auction, the First Price Sealed Bid (FPSB) auction, ... These auction protocols have different interaction sequences, but they all share the semantics of the basic communication actions, like *"bid"* and *"win"*. Communicative acts, such as *bid* and *answ*(*win*), can be defined in a general enough way, such that we can use the same acts in different protocols. IPs can then be seen, in this perspective, as additional sets of constraints, defining relations among communicative actions, which are to be added to those already defining the ACL, and which have to be consistent with them.

Let us consider the English auction protocol definition. In an English auction, the value of bids must be monotonically growing with time (11). Once a bid is made, either a higher bid is made (before a deadline), or the agent who made the bid wins the auction (12). Only one agent is the winner (13).[4]

[4]Indeed, since we provide a uniform syntax for protocol, ACL and properties specification, it then becomes a design choice to classify a certain ic_S as part of the semantic specification of a communication act or as a part of a protocol. If we consider the fact that only one agent is the "'winner" as an intrinsic property of the *win* communicative act, which should hold true for all protocols using *win*, then Eq. (13) could also be considered as defining the semantics of *win* together with Eq. (10).

The following ic_S can be used to define such a protocol:

$$\begin{array}{ll} & \mathbf{H}(tell(Bidder_1, Auc, bid(Item, Q_1)), T_1) \\ \rightarrow & \mathbf{EN}(tell(Bidder_2, Auc, bid(Item, Q_2)), T_2), T_2 > T_1, Q_2 \leq Q_1 \end{array} \tag{11}$$

$$\begin{array}{ll} & \mathbf{H}(tell(Auc, Bidders, opauc(Item, \tau, T_{notify}, english), D), T_{open}), \\ & \mathbf{H}(tell(Bidder_1, Auc, bid(Item, Q_1), D), T_1) \\ \rightarrow & \mathbf{E}(tell(Bidder_2, Auc, bid(Item, Q_2), D), T_2), \\ & Q_2 > Q_1, T_2 < T_1 + \tau \\ \vee & \mathbf{E}(tell(Auc, Bidder_1, answ(win, Item, Q_1), D), T_{win}), \\ & T_{win} < T_1 + T_{notify} \end{array} \tag{12}$$

$$\begin{array}{ll} & \mathbf{H}(tell(Auc, B_{win}, answ(win, Item, Q_{win}), D), T_{win}) \\ \rightarrow & \mathbf{EN}(tell(Auc, B_1, answ(win, Item, Q_1), D), T_Q), B_{win} \neq B_1 \end{array} \tag{13}$$

A different protocol is the First Price Sealed Bid (FPSB) auction protocol. In the FPSB auction, agents post their bids all during the same time interval, after the auction is declared open and before the deadline for bids has passed. The agent who makes the highest bid wins. The FPSB auction protocol can be defined by Eq. 13 together with the following ic_S:

$$\begin{array}{ll} & \mathbf{H}(tell(Auc, Bidders, opauc(Item, T_{dead}, T_{notify}, fpsb), D), T_{open}), \\ & \mathbf{H}(tell(Bidder_1, Auc, bid(Item, Q_1), D), T_1) \\ \rightarrow & \mathbf{E}(tell(Bidder_2, Auc, bid(Item, Q_2), D), T_2), \\ & Q_2 > Q_1, T_2 < T_{dead} \\ \vee & \mathbf{E}(tell(Auc, Bidder_1, answ(win, Item, Q_1), D), T_{win}), \\ & T_{win} < T_{dead} + T_{notify} \end{array} \tag{14}$$

By following an objective approach to interaction design, we detach the semantic specification of communicative acts and interaction protocols from the specification of agents. We can use *SOCS-SI* to test if the ic_S written as (9)-(14) do indeed produce sensible expectations or not. The availability of an implemented proof-procedure makes it possible to ground the validation of such a design step on a number of sample interaction traces, which can be kept as a collection of test cases or benchmarks, and parsed by the File System interface to the Event Recorder. The results of the runs of such sample traces can be visualized through the *SOCS-SI* GUI, and information shown on the tree-view can be used to possibly refine the existing ic_S or to add some new ones. A similar approach can be followed when validating the interaction space with respect to the properties that we would like to achieve by it, as we will discuss in the next section.

4.3 Properties definition and verification

An interaction space is "properly" designed if it exhibits some formally defined *properties*. For instance, in the design of an auction protocol, we would like to ensure that the agent who utters the highest bid will be assigned the goods at a certain price, and that it will pay for the price specified in its *bid*.

Following Pitt and Guerin [2002], the definition of properties, again, could follow a declarative and logic-based methodology, and in particular it could be done by means of integrity constraints. Let us consider the following example, adapted from [Pitt and Guerin, 2002].

The property informally stated above (the best bid wins) can be formally defined as follows: *"For all courses of events (***HAP***) such that there exists a consistent set of expectations (***EXP***), and such* **EXP** *is fulfilled by* **HAP** *(compliance condition), if there is a bid, then there is a winner"*:

$$\begin{array}{ll} & \mathbf{H}(tell(Auc, Bidder, opauc(Item, T_{dead}, T_{notify}, Type), D), T_{open}) \\ & \mathbf{H}(tell(Bidder_1, Auc, bid(Item, Q_1), D), T_1) \\ \rightarrow & \mathbf{E}(tell(Auc, Bidder_2, answ(win, Item, Q_2), D), T_{win}) \end{array} \tag{15}$$

Checking this property in this framework means in general considering all possible histories **HAP** complying with the ACLs and IPs, and checking whether property (15) is *entailed* by **HAP**.

Another property could be: *"for all compliant courses of events including a bid, there exists a deadline T_{dead} by which an agent is awarded to be the auction winner"*:

$$\begin{array}{ll} & \mathbf{H}(tell(Auc, Bidder, opauc(Item, T_{dead}, T_{notify}, Type), D), T_{open}) \\ & \mathbf{H}(tell(Bidder_1, Auc, bid(Item, Q_1), D), T_1) \\ \rightarrow & \mathbf{E}(tell(Auc, Bidder_2, answ(win, Item, Q_2), D), T_{win}), \\ & T_{win} < T_{dead} + T_{notify} \end{array} \tag{16}$$

Verifying this kind of entailment given a specific history instance is not hard, since all history instances are ground sets of facts. Therefore, this method is directly applicable when the set of possible histories of events is finite, or when we are interested in verifying properties only in some particular situations, which could be given a-priori. These can indeed be the sample traces used in the previous step for defining the interaction space itself.

During the verification of properties, it may turn out that under some circumstances a property is not achieved. In that case, this step or some previous ones should be iterated, and either the interaction space definition or the target properties refined.

For instance, it is easy to see that property (15) holds for both the English and the FPSB auction protocol, whereas it is possible to find a counterexample showing that property (16) does not hold for the the English auction protocol.

Once this turns out, if property (16) is considered important in the system that we are designing, an option could be to modify the English auction protocol, for instance by introducing a new ic_S. But, one could also reconsider the need for such a property, and realise that a less constraining property is enough.

Another situation that may occur is that some history instance produces only inconsistent sets of expectations (e.g., by producing both an expectation and its denial) but no violation. This is normally a sign that the interaction space has been ill-defined. Conversely, we also aim to avoid having history instances which we intend to be marked as inconsistent, but for which the $\mathcal{S}$CIFF generates a success node. This normally means that the interaction space is under-constrained, and again, we need to refine its specification.

5 Related work

The social approach to the definition of interaction protocols and semantics of Agent Communication Languages has been documented in several noteworthy contributions of the past years. Among them, Artikis et al. [2002] present a formal framework for specifying systems where the behaviour of the members and their interactions cannot be predicted in advance, and for reasoning about and verifying the properties of such systems. The framework relies upon a deontic logic formalism, and on the concepts of permission, prohibition, and empowerment. The paper also describes a *Society Visualiser* to demonstrate animations of protocol runs in such systems. A noteworthy difference with [Artikis *et al.*, 2002] is that we do not explicitly represent the institutional power of the members and the concept of valid action. "Permitted" are all social events that do not determine a violation, i.e., all events that are not explicitly "forbidden" are "allowed". Being detached from any deontic infrastructure, our framework can be used for a broader spectrum of application domains, from intelligent agents to reactive systems.

Huget [2004] proposes to describe goals through a graphical notation. Goals are composed of plans, which, in their turn, consist of actions. In a Goal Diagram the relationships among plans and actions are represented by means of nodes and arches. Goals can involve one or more agents. In PROSOCS, goals can be either goals of the single agent or of the society. Agent's goals can be decomposed into subgoals, in a tree-like hierarchical structure [Kakas *et al.*, 2004]. Society goals are given as a logical formula, that can contain literals defined in the society's knowledge base. The $\mathcal{S}$CIFF proof-procedure processes the goals and raises expectations about the agent's behaviour, that can be considered as the atomic actions expected by the society. Agents can then adopt the society's expectations as their own goals, or not.

From this viewpoint, our representation of social goals is more open, as it does not assume that agents will indeed comply to social expectations and adopt the society's goals.

Caire et al. [2004] propose an agent-oriented CASE tool for implementation and testing of Multi-Agent Systems. The testing framework is divided in two steps: the agent test and the society test. The agent test verifies the behaviour of the agent with regard to the system requirements under the responsibility of that agent; the agents are checked both in their black-box behaviour, and in a white-box checking of the behaviour of their internal modules. The "agent society testing is a kind of integration testing": the successful integration of the different agents is verified. The testing is performed automatically, without the need for intervention of the user.

Our work is devoted to testing on-the-fly the compliance of agents to social rules, without having any knowledge on the internals of the agents. We provided a language, based on logics, to define the interaction protocols, and a proof-procedure, based on abduction, to check the compliance. Our Society Infrastructure can be used to check the behaviour of Multi-Agent Systems that are *open*: members of the society are *not* only the ones defined by the MAS designer, but new agents, possibly malicious, may unpredictably join the society, and interact with the other agents. As far as their behaviour follows the society's prescriptions, such interactions may enrich the society, but they must be checked for conformance in order to avoid abuses.

Yolum and Singh [2002] apply a variant of Event Calculus [Kowalski and Sergot, 1986] to commitment-based protocol specification. The semantics of messages (i.e., their effect on commitments) is described by a set of *operations* whose semantics, in turn, is described by *predicates* on *events* and *fluents*; in addition, commitments can evolve, independently of communicative acts, in relation to *events* and *fluents* as prescribed by a set of *postulates*. Such a way of specifying protocols is more flexible than traditional approaches based on action sequences in that it prescribes no initial and final states or transitions explicitly. It only restricts the agent interaction in that, at the end of a protocol run, no commitment must be pending; agents with reasoning capabilities can themselves plan an execution path suitable for their purposes, by means of an Abductive Event Calculus planner. Our notion of expectation is more general than that of commitment adopted by Yolum and Singh [2002] or by other work, such as [Fornara and Colombetti, 2002]: it represents the expectation about a (past or future) event, without any reference to specific roles of agents (such as a commitment's debtor and creditor), and it does not necessarily need to be brought about by a specific agent.

Finally, several other frameworks in the literature aim at verifying properties about the behaviour of social agents at design time. Often, such frameworks define structured hierarchies, roles, and deontic concepts such as norms and obligations as first class entities. Notably, ISLANDER [Esteva *et al.*, 2002] is a tool for the

specification and verification of interaction in complex social infrastructures, such as electronic institutions. ISLANDER allows to analyse situations, called scenes, and visualise liveness or safeness properties in some specific settings. The kind of verification involved is static and is used to help designing institutions. Although our framework could also be used at design time, its main intended use is for on-the-fly verification of heterogenous and open systems.

6 Conclusions and future work

The study and specification of interaction has since long been a central topic of agent research. In recent years, considerable effort has been devoted to developing both methodologies and tools for the verification of agent interaction, and languages and formalisms for the specification of agent communication languages and interaction protocols, a domain which seems to be well suited for formal approaches.

The purpose of this work is to propose a formal framework and a methodology, encompassing together specification and verification of agent communication languages and interaction protocols. The methodology is a property-driven cyclic process interleaving specification and refinement steps with verification steps. In support to such a process, we propose and describe in this paper a software component for the verification of compliance of agent interaction to a specification given in a logic-based formalism. The component, called *SOCS-SI*, implements the verification procedure using the CHR library of SICStus Prolog, and it features a graphical user interface and multiple input sources. Its use within the methodology has been demonstrated with an auction protocol design example. To the best of our knowledge this is the first work in which a fully implemented operational social framework is presented, aimed at the automatic verification of agent interaction. Building on a formal ground, our work contributes towards bridging the gap between theory and implementation of multi-agent systems.

Future work will be devoted to studying properties of agent interaction at runtime and at design time, in combination with PROSOCS, to refining the interaction design methodology especially by analysing the possible connections between *SOCS-SI* and other existing agent-oriented software engineering approaches, and to interfacing *SOCS-SI* with other existing agent platforms, such as JADE [Bellifemine *et al.*, 2000], so to act as a further verification layer. Implementation-wise, it would be interesting to study the possibility to have the proof-procedure distributed in several nodes of a networked environment, each devoted to observing only a subset of the whole interactions occurring within the network. Finally, we would like to investigate two further applications of *SOCS-SI*: (*i*) to support the generation and management of agent reputation and trust, by discriminating complying agents from those behaving in unexpected ways, and (*ii*) as a facilitator for agents entering new societies, by feeding agents with expectations and therefore providing them with

knowledge about the society protocols.

Acknowledgments

We thank the anonymous referees for their valuable suggestions which greatly helped improve this paper.

This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-32530 SOCS project [SOCS, 2001], within the Global Computing proactive initiative, and by the national MIUR COFIN 2003 projects "Sviluppo e verifica di sistemi nulti-agente basati sulla logica" and "La gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici".

References

[Alberti *et al.*, 2003a] M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. A social ACL semantics by deontic constraints. In V. Mařík, J. Müller, and M. Pěchouček, editors, *Multi-Agent Systems and Applications III. Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003*, volume 2691 of *Lecture Notes in Artificial Intelligence*, pages 204–213, Prague, Czech Republic, June 16-18 2003. Springer-Verlag.

[Alberti *et al.*, 2003b] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. An Abductive Interpretation for Open Societies. In A. Cappelli and F. Turini, editors, *AI*IA 2003: Advances in Artificial Intelligence, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa*, volume 2829 of *Lecture Notes in Artificial Intelligence*, pages 287–299. Springer-Verlag, September 23–26 2003.

[Alberti *et al.*, 2003c] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity constraints. *Electronic Notes in Theoretical Computer Science*, 85(2), 2003.

[Alberti *et al.*, 2003d] Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Specification and verification of interaction protocols: a computational logic approach based on abduction. Technical Report CS-2003-03, Dipartimento di Ingegneria di Ferrara, Ferrara, Italy, 2003. `http://www.ing.unife.it/informatica/tr/`.

[Artikis *et al.*, 2002] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and W. Lewis Johnson, editors,

Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III, pages 1053–1061, Bologna, Italy, July 15–19 2002. ACM Press.

[Bellifemine *et al.*, 2000] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with JADE. In Cristiano Castelfranchi and Yves Lespérance, editors, *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop, ATAL 2000, Boston, MA, USA, July 7-9, 2000, Proceedings*, volume 1986 of *Lecture Notes in Computer Science*, pages 89–103. Springer Verlag, 2000.

[Caire *et al.*, 2004] G. Caire, M. Cossentino, A. Negri, A. Poggi, and P. Turci. Multi-agent systems implementation and testing. In Robert Trappl, editor, *Cybernetics and Systems 2004 - Volume II*, pages 612–617, Vienna, Austria, April 13 - 16 2004. Austrian Society for Cybernetics Studies.

[Davidsson, 2001] P. Davidsson. Categories of artificial societies. In A. Omicini, P. Petta, and R. Tolksdorf, editors, *Engineering Societies in the Agents World II*, volume 2203 of *Lecture Notes in Artificial Intelligence*, pages 1–9. Springer-Verlag, December 2001.

[d'Inverno *et al.*, 1998] M. d'Inverno, D. Kinny, and M. Luck. Interaction protocols in Agentis. In *Proceedings of the 3rd International Conference on Multiagent Systems, San Francisco, California*, pages 261–268, 1998.

[Esteva *et al.*, 2002] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III*, pages 1045–1052, Bologna, Italy, July 15–19 2002. ACM Press.

[Fornara and Colombetti, 2002] N. Fornara and M. Colombetti. Operational specification of a commitment-based agent communication language. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 535–542, Bologna, Italy, July 15–19 2002. ACM Press.

[Fornara and Colombetti, 2003] N. Fornara and M. Colombetti. Defining interaction protocols using a commitment-based agent communication language. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 520–527, Melbourne, Victoria, July 14–18 2003. ACM Press.

[Frühwirth, 1998] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

[Fung and Kowalski, 1997] T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, November 1997.

[Gelernter and Carriero, 1992] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Huget, 2004] Marc-Philippe Huget. Representing goals in multiagent systems. In Robert Trappl, editor, *Cybernetics and Systems 2004 - Volume II*, pages 588–593, Vienna, Austria, April 13 - 16 2004. Austrian Society for Cybernetics Studies.

[Jaffar and Maher, 1994] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[Kakas *et al.*, 1993] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive Logic Programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[Kakas *et al.*, 2004] A.C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In R. Lopez de Mantaras and L. Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence*. IOS Press, August 2004. to appear.

[Kinny *et al.*, 1996] D. Kinny, M. Georgeff, and A. S. Rao. A methodology and modelling technique for systems of BDI agents. In Rudy van Hoe, editor, *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer-Verlag, 1996.

[Kowalski and Sergot, 1986] R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[Omicini and Ossowski, 2003] Andrea Omicini and Sascha Ossowski. Objective versus subjective coordination in the engineering of agent systems. In Matthias Klusch, Sonia Bergamaschi, Peter Edwards, and Paolo Petta, editors, *Intelligent Information Agents: An AgentLink Perspective*, volume 2586 of *LNAI: State-of-the-Art Survey*, pages 179–202. Springer-Verlag, March 2003.

[Pitt and Guerin, 2002] J. Pitt and F. Guerin. Guaranteeing properties for e-commerce systems. Technical Report TRS020015, Department of Electrical and

Electronic Engineering, Imperial College, London, UK, 2002. http://www.iis.ee.ic.ac.uk/reports.

[Ricci *et al.*, 2002] Alessandro Ricci, Andrea Omicini, and Enrico Denti. Objective vs. subjective coordination in agent-based systems: A case study. In Farhad Arbab and Carolyn Talcott, editors, *Coordination Languages and Models*, volume 2315 of *LNCS*, pages 291–299. Springer-Verlag, 2002. 5th International Conference (COORDINATION 2002), York, UK, 8–11 April 2002. Proceedings.

[SICStus, 2003] SICStus prolog user manual, release 3.11.0, October 2003. http://www.sics.se/isl/sicstus/.

[SOCS, 2001] SOCS. Societies Of ComputeeS: a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees, 2001. http://lia.deis.unibo.it/Research/SOCS/.

[Stathis *et al.*, 2004] Kostas Stathis, Antonis C. Kakas, Wenjin Lu, Neophytos Demetriou, Ulle Endriss, and Andrea Bracciali. PROSOCS: a platform for programming software agents in computational logic. pages 523–528, Vienna, Austria, April 13-16 2004. Austrian Society for Cybernetic Studies.

[Torroni *et al.*, 2004] Paolo Torroni, Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, and Paola Mello. A logic based approach to interaction design in open multi-agent systems. In *Proceedings of the 13th IEEE international Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), 2nd international workshop "Theory and Practice of Open Computational Systems (TAPOCS)"*, Modena, Italy, June 14 2004. IEEE Press. to appear.

[Verdicchio and Colombetti, 2003] M. Verdicchio and M. Colombetti. A logical model of social commitment for agent communication. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 528–535, Melbourne, Victoria, July 14–18 2003. ACM Press.

[Wooldridge *et al.*, 2000] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[Yolum and Singh, 2002] P. Yolum and M.P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II*, pages 527–534, Bologna, Italy, July 15–19 2002. ACM Press.