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

Marco Alberti Marco Gavanelli Evelina Lamma
DI, Università di Ferrara
Via Saragat, 1
44100 Ferrara, Italy
{malberti, mgavanelli, elamma}@ing.unife.it

Federico Chesani Paola Mello Paolo Torroni
DEIS, Università di Bologna
V.le Risorgimento, 2
40136 Bologna, Italy
{fchesani,pmello,ptorroni}@deis.unibo.it

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 which can be used to verify compliance of agent interaction to protocols written in a logic-based formalism (*Social Integrity Constraints*).

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 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 agent *observable* behaviour rather than 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], Fornara and Colombetti [2003], Verdicchio and Colombetti [2003] and Artikis et. al. [2002].

In previous work [Alberti *et al.*, 2003c], we proposed 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 a tool (*SOCS-SI*) that is a Java-Prolog-CHR based implementation of the proof-procedure defined in [Alberti *et al.*, 2003d]. The intended use of *SOCS-SI*[1] 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. 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,

[1] "*SOCS*" is the acronym of the EU-funded project (IST-2001-32530) that partially supported this work [SOCS, 2002]. *SI* stands for *Social Infrastructure.*

out work contributes towards bridging the gap between theory and implementation of multi-agent systems.

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. Discussion of related work and conclusions follow.

2 Logic-based Specification and Verification

In this section we give the necessary background on the formal framework proposed by Alberti et al. [2003a; 2003c; 2003b] for the specification of agent interaction in open[2] societies of agents. The reader is referred to those papers for a complete description.

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, $\mathbf{H}(\mathit{request}(a_i, a_j, \mathit{give}(10\$), d_1), 7)$ 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 7.[3]

In open agent societies, the agent behaviour is unpredictable, because agents are autonomous; however, when interaction protocols are defined, we are able to determine what are the possible expectations about future events. This represents 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. CLP [Jaffar and Maher, 1994] constraints can be imposed on variables to restrict their domain.

For instance,
$\mathbf{E}(\mathit{accept}(a_k, a_j, \mathit{giv}(M), d_2), T_{\mathrm{a}}) : M \geq 10, T_{\mathrm{a}} \leq 15$
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 less or equal than 15.

The way expectations are generated, given the happened events and the current expectations, is specified by means of *Social Integrity Constraints* (IC_S).

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

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

states that, if agent A makes a *request* of 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:

$$\begin{array}{l} \mathbf{H}(\mathit{accept}(A,B,P,D),T_1) \\ \rightarrow \mathbf{EN}(\mathit{refuse}(A,B,P,D),T_2) : T_2 \geq T_1 \end{array} \quad (2)$$

$$\begin{array}{l} \mathbf{H}(\mathit{refuse}(A,B,P,D),T_1) \\ \rightarrow \mathbf{EN}(\mathit{accept}(A,B,P,D),T_2) : T_2 \geq T_1 \end{array} \quad (3)$$

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.

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 formalize expectations about agent behaviour as abducibles, and to use Social Integrity Constraints such as (1), (2) or (3) to prevent such agent behaviour that is not compliant with interaction protocols.

Given the partial history of a society, an abductive proof procedure (SCIFF, [Alberti *et al.*, 2003d]) generates expectations about agent behaviour so as to comply with Social Integrity Constraints. SCIFF is inspired by the IFF proof procedure, [Fung and Kowalski, 1997] augmented as needed to manage CLP constraints. The most distinctive feature of SCIFF, however, is its ability to

[2] We intend *openness* in societies of agents as Artikis et al. [2002], where agents can be heterogeneous and possibly non-cooperative.

[3] 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.

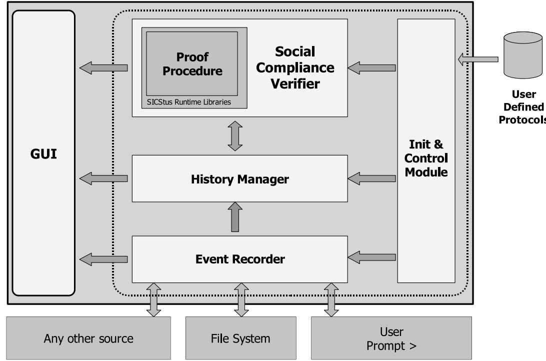

Figure 1: Overview of the *SOCS-SI* architecture

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.

3 The *SOCS-SI* Tool

In this section, we describe the implementation of the *SOCS-SI* tool for compliance verification of agent interaction. The tool 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 by a set of modules. All the components except one (the proof-procedure) are implemented in the Java language.

The core of *SOCS-SI* is composed by 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 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 initialization 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.

3.1 Implementation of the proof-procedure

For the implementation of the society proof-procedure, SICStus Prolog [SICStus, 2000] has been chosen, for the following reasons:

- the Prolog language offers built-in facilities for the implementation of dynamic data structures and (customizable) search strategies;
- SICStus Prolog allows for state-of-the-art CLP; in particular, the CLPB, CLPFD and CHR libraries 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 social proof-procedure described in [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.

The Prolog-CHR module implements the transitions of the proof procedure. CHR [Frühwirth, 1998] is a rewriting system for implementing new constraints. It is based on forward rules that rewrite constraints into other constraints. By implementing the data structures of the proof procedure (eg. PSIC, EXP) as CHR constraints, the transitions can be implemented as CHR rules.

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 ac-

cessed 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 initialization time. Our software application can deal with different proof-procedure implementations, 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 by 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 specialized for a specific source. For testing and debugging purposes, we developed modules to interact with the user prompt as well as with the file system; it is possible to add as many specialized 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 (through the configuration GUI, or modifying the config file).

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

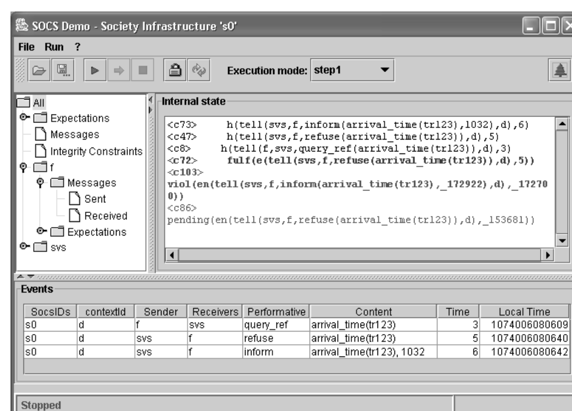

Figure 2: A screenshot of the application

- `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 actual 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, 2002]. 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-window), and of a button bar that contains the controls (see Fig. 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 emphasized (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. 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 behavior 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).

4 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 Visualizer* 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.

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 analyze situations, called scenes, and visualize 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.

5 Conclusions and future work

In current years, there is a considerable ongoing effort in the area of interaction specification, and the domain seems to be well suited for formal approaches. In this paper, we presented a software component for the verification of compliance of agent interaction to a specification given in a logic-based formalism. The component features a graphical user interface, and can receive input from several possible sources. The portion of the software devoted to verification of compliance implements an abductive proof-procedure by using SICStus Prolog and, in particular, its CHR library.

Future work will be devoted to studying properties of agent interaction at runtime and at design time, in combination with PROSOCS, to developing a methodology for designing the interaction space of multi-agent systems (protocols and semantics of communicative acts), by using *SOCS-SI* possibly in combination with other existing methodologies, and to interfacing the component to other existing agent platforms such as JADE [JADE, 2000], so to act as a further ver-

ification layer. It would be interesting to study the possibility to distribute the proof-procedure, so that some partial abducibles can be abduced on separated nodes of a networked environment, observing only a subset of an interaction. 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 is partially funded by the IST programme of the European Commission under the IST-2001-32530 SOCS project [SOCS, 2002], 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 *Multi-Agent Systems and Applications III*, *Lecture Notes in Artificial Intelligence* 2691, pages 204–213. Springer-Verlag.

[Alberti *et al.*, 2003b] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. An Abductive Interpretation for Open Societies. In *AI*IA 2003: Advances in Artificial Intelligence*, *Lecture Notes in Artificial Intelligence* 2829, pages 287–299. Springer-Verlag.

[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] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. 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.

[Artikis *et al.*, 2002] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In *Proc. AAMAS-2002, Part III*, pages 1053–1061. ACM Press.

[Esteva *et al.*, 2002] M. Esteva, D. de la Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. In *Proc. AAMAS-2002, Part III*, pages 1045–1052. ACM Press.

[Fornara and Colombetti, 2002] N. Fornara and M. Colombetti. Operational specification of a commitment-based agent communication language. In *Proc. AAMAS-2002, Part II*, pages 535–542. ACM Press.

[Fornara and Colombetti, 2003] N. Fornara and M. Colombetti. Defining interaction protocols using a commitment-based agent communication language. In *Proc. AAMAS-2003*, pages 520–527. 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.

[JADE, 2000] Java Agent DEvelopment framework. Home Page: `http://sharon.cselt.it/projects/jade/`.

[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.

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

[SICStus, 2000] SICStus prolog user manual, release 3.8.4, May 2000. `http://www.sics.se/isl/sicstus/`.

[SOCS, 2002] Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. `http://lia.deis.unibo.it/Research/SOCS/`.

[Stathis *et al.*, 2004] K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: a platform for programming software agents in computational logic. In this volume.

[Verdicchio and Colombetti, 2003] M. Verdicchio and M. Colombetti. A logical model of social commitment for agent communication. In *Proc. AAMAS-2003*, pages 528–535. ACM Press.

[Yolum and Singh, 2002] P. Yolum and M. P. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In *Proc. AAMAS-2002, Part II*, pages 527–534. ACM Press.