Scalable Computing: Practice and Experience
Volume 8, Number 1, pp. 1–13. http://www.scpe.org
ISSN 1895-1767

SPECIFICATION AND VERIFICATION OF AGENT INTERACTION PROTOCOLS IN A LOGIC-BASED SYSTEM*

MARCO ALBERTI, FEDERICO CHESANI, DAVIDE DAOLIO, MARCO GAVANELLI, EVELINA LAMMA, PAOLA MELLO AND PAOLO TORRONI

Abstract.
A number of information systems can be described as a set of interacting entities, which must follow interaction protocols. These protocols determine the behaviour and the properties of the overall system, hence it is of the uttermost importance that the entities behave in a conformant manner.

A typical case is that of multi-agent systems, composed of a plurality of agents without a centralized control. Compliance to protocols can be hardwired in agent programs; however, this requires that only "certified" agents interact. In open systems, composed of autonomous and heterogeneous entities whose internal structure is, in general, not accessible (open agent societies being, again, a prominent example) interaction protocols should be specified in terms of the *observable* behaviour, and compliance should be verified by an external entity.

In this paper, we propose a Java-Prolog-*CHR* system for verification of compliance of computational entities to protocols specified in a logic-based formalism (*Social Integrity Constraints*). We also show the application of the formalism and the system to the specification and verification of three different scenarios: two specifications show the feasibility of our approach in the context of Multi Agent Systems (FIPA Contract-Net Protocol and Semi-Open societies), while a third specification applies to the specification of a lower level protocol (Open-Connection phase of the TCP protocol).

1. Introduction. Many information systems can be described as a set of mutually independent, interacting entities. A typical example is that of multi-agent systems. In such a scenario the interaction is usually subject to some kind of interaction protocols, which the agents should respect when interacting. This raises the obvious problem of verifying that interaction protocols are actually followed.

It is possible to design agents so that they will "spontaneously" comply to protocols, and, if possible, formally verify that at design time. For instance, in [13], Endriss et al. propose an approach where protocols are "imported" into individual agent policies.

However, this approach is not viable in open[1] agent societies, where interacting agents are autonomous and heterogeneous and, in general, their internal structure cannot be accessed. In this case, agents should be checked for compliance to interaction protocols based on their *observable* behaviour, by a trusted external entity.

In previous work [5], we proposed a computational logic-based formalism (based upon *Social Integrity Constraints*, SICs) to specify interaction protocols. Social Integrity Constraints are meant to constrain the agent observable behaviour rather than agents' internal (mental) state or policies. In other words, this approach does not restrict an agent's access to societies based on its internal structure; regardless of its policies, any agent can successfully interact in a society ruled by SICs, as long as its behaviour is compliant. The formal semantics of Social Integrity Constraints [4] is based on abductive logic programming [18].

The purpose of this paper is to demonstrate the viability of Social Integrity Constraints as a formalism to specify interaction between computational entities, including, but not limited to, agents in open societies. We will use a modified version of Social Integrity Constraints, which better fits our needs in terms of both simplicity of presentation, and expressiveness.

The paper is structured as follows. In Sect. 2, we introduce the version of Social Integrity Constraints used in this work, giving their syntax and an informal explanation of their semantics.

In Sect. 3 we specify in terms of SICs a contract net-based protocol for resource allocation and negotiation in multi-agent systems, called FIPA CNP, and in Sect. 4 we specify a protocol for entering "semi-open" societies, i. e., virtual environments characterized by the presence of a "gatekeeper" agent and a protocol that governs the agents' access to the society. In Sect. 5 we demonstrate the usage of SICs to specify a network communication protocol, namely the three-way handshake opening of the TCP Internet Protocol.

The article ends with the presentation of the compliance verification system (Sect. 6), and some notes about its Java+Prolog implementation.

*This article is an extended version of the one by Alberti, Daolio, Gavanelli, Lamma, Mello, and Torroni, published in Haddad, Omicini, and Wainwright, eds., *Proceedings of the 19th ACM Symposium on Applied Computing, SAC 2004, Special Track on Agents, Interactions, Mobility, and Systems (AIMS). Nicosa, Cyprus, March 14-17, 2004.* pp. 72-78. ACM Press (2004).

[1]We intend *openness* in societies of agents as Artikis, Pitt and Sergot [7], where agents can be heterogeneous and possibly non-cooperative.

2. Social Integrity Constraints. We distinguish between actual behaviour (*happened events*) and desired behaviour (*expectations*), since in non-ideal situations they do not always coincide. In this section, we let the reader get acquainted with our representation of events and we introduce Social Integrity Constraints (SICs) as a formalism used to express which expectations are generated as consequence of happened events.

Happened Events and Expectations. Happened events are in the form

$$\mathbf{H}(\mathit{Description}, \mathit{Time})$$

where *Description* is a term (as intended in logic programming, see [20]) representing the event that has happened, and *Time* is an integer number representing the time at which the event has happened. 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.

All happened events form the history of a society. Given the history of a society at a given time, some events will have to happen in order for interaction protocols to be satisfied: we represent such events by means of *expectations*, which can be *positive* or *negative*. Positive expectations are of the form

$$\mathbf{E}(\mathit{Description}, \mathit{Time})$$

and represent an event that is expected to happen (typically, an action that an agent is expected to take). Negative expectations are of the form

$$\mathbf{EN}(\mathit{Description}, \mathit{Time})$$

and represent the fact that an event is expected *not* to happen.

Expectations may (and, typically, will) contain variables, to reflect the fact that the expected event is not fully specified; however, CLP [17] constraints can be imposed on variables to restrict their domain. For instance,

$$\mathbf{E}(\mathit{accept}(a_k, a_j, \mathit{give}(M), d_2), T_\mathrm{a}) : M \geq 10, T_\mathrm{a} \leq 15 \tag{2.1}$$

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 at time T_a; moreover, M is expected to be at least 10\$, and T_a to be at most 15.

Since we impose no restrictions on the *Description* term of an expectation, expectations can regard any kind of event that can be expressed by a Prolog-like term. However, expectations only regard point-time events; thus it is not possible to express concisely that some *proposition* is expected to be true along a given time interval.

Since we make no assumptions about the agents' internal structure or policies, their behaviour may or may not satisfy expectations. We represent these two cases by means of the notions of *fulfillment* and *violation*. We say that an event *matches* an expectation if and only if:

- their contents unify (à la Prolog);
- all relevant CLP constraints on variables (if any) are satisfied.

A positive expectation can get *fulfilled* by a matching event, whereas a negative expectation can get *violated* by a matching event.

For instance, event

$$\mathbf{H}(\mathit{accept}(a_k, a_j, \mathit{give}(20), d_2), 15)$$

fulfills expectation (2.1); the same event would, instead, violate a negative expectations with the same content and CLP constraints.

If we assume at some point that no more events will ever occur, we say that the history is *closed*. In that case, all positive expectations that are not fulfilled are violated, and all negative expectations that are not violated are fulfilled.

TABLE 2.1
BNF syntax of Social Integrity Constraints

SIC	::= $\chi \rightarrow \phi$
χ	::= EventLiteral [$\wedge$ EventLiteral]* [:CList]
ϕ	::= PriorityLevel [$\Rightarrow$ PriorityLevel]*
PriorityLevel	::= HeadDisjunct [$\vee$ HeadDisjunct]*, P
EventLiteral	::= H(Term,T)
HeadDisjunct	::= Expectation [$\wedge$ Expectation]* [:CList]
Expectation	::= E(Term,T) | EN(Term,T)

Social Integrity Constraints. The way expectations are generated, given a (partial) history of a society, is specified by *Social Integrity Constraints* (SICs). In this article, we adopt a modified version of the SICs introduced in [2] (we discuss and motivate such modifications in Sect. 7).

Table 2.1 reports the BNF syntax of SICs. *Term* is a logic programming term [20], P is an integer number and T is a variable symbol or integer number. *CList* is a conjunction of CLP constraints on variables.

SICs are a kind of forward rules, stating what expectations should be generated on the basis of happened events. By means of SICs, it is possible to express that conjunctions of expectations (*HeadDisjuncts* in Table 2.1) are alternative, and it is also possible to assign a priority, represented by an integer number, to each list of alternatives (*PriorityLevel*s in Table 2.1).

For instance, the following SIC:

$$\begin{aligned} &\mathbf{H}(e_0, T_0) \wedge \mathbf{H}(e_1, T_1) : T_0 < T_1 \\ &\rightarrow \mathbf{E}(e_2, T_2) : T_2 < T_1 \vee \mathbf{EN}(e_3, T_3) : T_3 < T_0, 1 \\ &\Rightarrow \mathbf{E}(e_4, T_4) : T_4 < T_0, 2 \end{aligned} \tag{2.2}$$

means that, if e_0 happens before e_1, then either of the two cases below hold:

- e_2 should have happened before e_1 or e_3 should not have happened before e_0,
- e_4 should have happened before e_0;

and the first case has higher priority than (or is preferred to) the second one. Intuitively, a SIC means that, when a set of events matching its body happens, then at least one of the "priority levels" in its conclusion should be satisfied (the higher the priority, the better). In this case, we say that the SIC is *fulfilled*; otherwise, it is *violated*. While priorities have no effect upon the fulfillment status of the society, they could instead be used by a possible computational entity representing the society to guide its members' behaviour towards some preferred state. This can be useful when expectations are accounted for by agents deliberating about future actions. At each point in time there are in general several equally fulfilled sets of expectations. But if some are more preferred to others, an imaginary "social reasoner" which produces expectations based on events could then evaluate and choose which sets of expectations better fit its goals, and transmit only them to the society members. If such members take expectations into account, the whole society could evolve towards preferred states.

The expectations in SIC (2.2) regard events that should have (or have not) happened before the time of the event that raises them: we call this kind of expectations *backward*. Expectations that regard events that are expected to happen (or not to happen) after the event that raises them are named *forward*. We restrict the possible SICs by requiring that they contain only either backward expectations or forward expectations: in the first case, we will call the SIC *backward*, in the second case *forward*. We discuss this restriction in Sect. 7.

3. Specification of the FIPA Contract-Net. FIPA-CNP [1] is a protocol based on FIPA-ACL [14] defined for regulating transactions between entities by negotiation. The protocol flow, represented as an AUML [21] diagram in Fig. 3.1, starts with an Initiator which issues a request for a resource (*cfp*, standing for *call for proposals*) to other Participants. The Participants can reply by proposing a price that satisfies the request (*propose*), or by refusing the request altogether (*refuse*). The Initiator must accept (*accept-proposal*) or reject (*reject-proposal*) the received proposals. A Participant whose proposal has been accepted must, by a given deadline, inform the Initiator that it has provided the resource (by sending an *inform-done* message, or a more informative *inform-result* message) or that it has failed to provide it (*failure*).

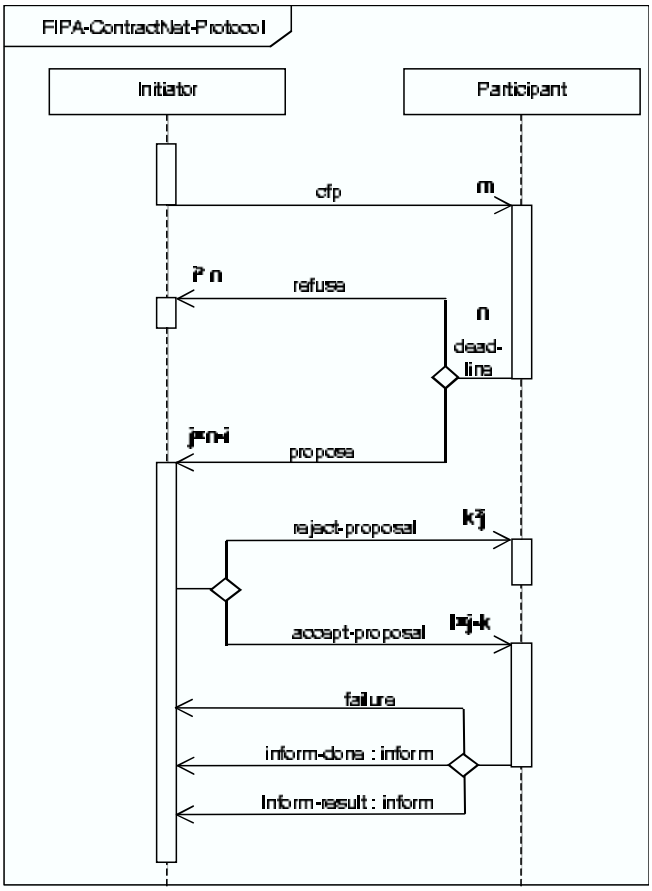

FIG. 3.1. *FIPA-Contract-Net Interaction Protocol (AUML Diagram)*

3.1. Definition by Social Integrity Constraints. The whole set of SICs used to define FIPA-CN is composed of 14 backward SICs and 3 forward SICs. This choice of SICs is obviously not the only possibility. We are currently investigating a general mapping of AUML protocol diagrams and other graphical formalisms to SICs, so as to allow for an automatic translation. Some progress in this sense has been done with the GOSpel graphic language [10] in the health care application domain.

In the SICs in the remainder of this section, I will represent the initiator, P a participant, R the resource, Q the price, D the dialogue identifier, S the explanation of a result, and $T, T_1, \ldots$ the time. We will not use priority levels.

Backward SICs. Backward SICs are used to express that an action is only allowed if some other events have (not) occurred before.

SICs (3.1) and (3.2) state that *propose* and *refuse* are only allowed in reply to a *cfp*.

$$\begin{aligned}&\mathbf{H}(tell(P, I, propose(R, Q), D), T) \rightarrow \\ &\mathbf{E}(tell(I, P, cfp(R), D), T_1) : T_1 < T\end{aligned} \tag{3.1}$$

$$\begin{aligned}&\mathbf{H}(tell(P, I, refuse(R), D), T) \rightarrow \\ &\mathbf{E}(tell(I, P, cfp(R), D), T_1) : T_1 < T\end{aligned} \tag{3.2}$$

SICs (3.3) and (3.4) express mutual exclusion between *propose* and *refuse*.

$$\begin{aligned}&\mathbf{H}(tell(P, I, propose(R, Q), D), T) \rightarrow \\ &\mathbf{EN}(tell(P, I, refuse(R), D), T_1) : T_1 \leq T\end{aligned} \tag{3.3}$$

$$\begin{aligned}&\mathbf{H}(tell(P, I, refuse(R), D), T) \rightarrow \\ &\mathbf{EN}(tell(P, I, propose(R, Q), D), T_1) : T_1 \leq T\end{aligned} \tag{3.4}$$

SICs (3.5) and (3.6) state that *accept-proposal* and *reject-proposal* are only allowed in reply to a *propose*.

$$\begin{aligned}&\mathbf{H}(tell(I, P, accept\text{-}proposal(R, Q), D), T) \rightarrow \\ &\mathbf{E}(tell(P, I, propose(R, Q), D), T_1) : T_1 < T\end{aligned} \tag{3.5}$$

$$\begin{aligned}&\mathbf{H}(tell(I, P, reject\text{-}proposal(R, Q), D), T) \rightarrow \\ &\mathbf{E}(tell(P, I, propose(R, Q), D), T_1) : T_1 < T\end{aligned} \tag{3.6}$$

SICs (3.7) and (3.8) express mutual exclusion between *accept-proposal* and *reject-proposal*.

$$\begin{aligned}&\mathbf{H}(tell(I, P, accept\text{-}proposal(R, Q), D), T) \rightarrow \\ &\mathbf{EN}(tell(I, P, reject\text{-}proposal(R, Q), D), T_1) : T_1 \leq T\end{aligned} \tag{3.7}$$

$$\begin{aligned}&\mathbf{H}(tell(I, P, reject\text{-}proposal(R, Q), D), T) \rightarrow \\ &\mathbf{EN}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 \leq T\end{aligned} \tag{3.8}$$

SICs (3.9), (3.10) and (3.11) say that *inform-done*, *inform-result* and *failure* are only allowed in reply to an *accept-proposal*.

$$\begin{aligned}&\mathbf{H}(tell(P, I, inform\text{-}done(R), D), T) \rightarrow \\ &\mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T\end{aligned} \tag{3.9}$$

$$\begin{aligned}&\mathbf{H}(tell(P, I, inform\text{-}result(R, S), D), T) \rightarrow \\ &\mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T\end{aligned} \tag{3.10}$$

$$\begin{aligned}&\mathbf{H}(tell(P, I, failure(R), D), T) \rightarrow \\ &\mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T\end{aligned} \tag{3.11}$$

SICs (3.12), (3.13) and (3.14) express mutual exclusion between *inform-done*, *inform-result* and *failure*.

$$\begin{aligned}&\mathbf{H}(tell(P, I, inform\text{-}done(R), D), T) \rightarrow \\ &\mathbf{EN}(tell(P, I, failure(R), D), T_1) : T_1 \leq T \wedge \\ &\mathbf{EN}(tell(P, I, inform\text{-}result(R, S), D), T_1) : T_1 \leq T\end{aligned} \tag{3.12}$$

$$\begin{aligned}&\mathbf{H}(tell(P, I, inform\text{-}result(R, S), D), T) \rightarrow \\ &\mathbf{EN}(tell(P, I, failure(R), D), T_1) : T_1 \leq T \wedge \\ &\mathbf{EN}(tell(P, I, inform\text{-}done(R), D), T_1) : T_1 \leq T\end{aligned} \tag{3.13}$$

$$\begin{aligned}&\mathbf{H}(tell(P, I, failure(R), D), T) \rightarrow \\ &\mathbf{EN}(tell(P, I, inform\text{-}done(R), D), T_1) : T_1 \leq T \wedge \\ &\mathbf{EN}(tell(P, I, inform\text{-}result(R, S), D), T_1) : T_1 \leq T\end{aligned} \tag{3.14}$$

Forward SICs. SIC (3.15) says that, after receiving a *cfp*, a Participant is expected to issue a *propose* or a *refuse* by 200 time units.[2]

$$\begin{aligned}&\mathbf{H}(tell(I,P,cfp(R),D),T)\rightarrow\\&\mathbf{E}(tell(P,I,propose(R,Q),D),T_1):T_1<T+200\vee\\&\mathbf{E}(tell(P,I,refuse(R),D),T_2):T_2<T+200\end{aligned}\tag{3.15}$$

SIC (3.16) states that the Initiator is expected to reply to a *propose* with an *accept-proposal* or a *reject-proposal* by 200 clock ticks.

$$\begin{aligned}&\mathbf{H}(tell(P,I,propose(R,Q),D),T)\rightarrow\\&\mathbf{E}(tell(I,P,accept\text{-}proposal(R,Q),D),T_1):T_1<T+200\vee\\&\mathbf{E}(tell(I,P,reject\text{-}proposal(R,Q),D),T_2):T_2<T+200\end{aligned}\tag{3.16}$$

SIC (3.17) states that a Participant is expected to reply to an *accept-proposal* with an *inform-done*, an *inform-result* or a *failure* by 200 clock ticks.

$$\begin{aligned}&\mathbf{H}(tell(I,P,accept\text{-}proposal(R,Q),D),T)\rightarrow\\&\mathbf{E}(tell(P,I,inform\text{-}done(R),D),T_1):T_1<T+200\vee\\&\mathbf{E}(tell(P,I,inform\text{-}result(R,S),D),T_2):T_2<T+200\vee\\&\mathbf{E}(tell(P,I,failure(R),D),T_2):T_2<T+200\end{aligned}\tag{3.17}$$

Note that, in all the three cases, backward SICs make the alternative expectations mutually exclusive.

4. Specification of a semi-open society access protocol. According to [11], societies can be classified into 4 groups, each characterized by a different degree of openness. In the following, we give an example of how our framework can model a semi-open society, i. e., a society that can be joined by an agent executing an access protocol. In this example we imagine that a special *gatekeeper* agent is in charge of receiving joining requests, and it requests agents willing to enter to fill in some registration form.

The access protocol is defined by the following SICs, in which C represents the name of an agent willing to join in:

$$\begin{aligned}&\mathbf{H}(tell(C,gatekeeper,ask(register),D),T)\rightarrow\\&\mathbf{E}(tell(gatekeeper,C,ask(form),D),T_1):T_1<T+10\end{aligned}\tag{4.1}$$

$$\begin{aligned}&\mathbf{H}(tell(C,gatekeeper,ask(register),D),T)\wedge\\&\mathbf{H}(tell(gatekeeper,C,ask(form),D),T_1)\wedge T<T_1\rightarrow\\&\mathbf{E}(tell(C,gatekeeper,send(form,F),D),T_2):T_2<T_1+10\end{aligned}\tag{4.2}$$

$$\begin{aligned}&\mathbf{H}(tell(gatekeeper,C,ask(form),D),T_1)\wedge\\&\mathbf{H}(tell(C,gatekeeper,send(form,F),D),T_2)\wedge T_1<T_2\rightarrow\\&\mathbf{E}(tell(gatekeeper,C,accept(register),D),T_3):T_3<T_2+10\vee\\&\mathbf{E}(tell(gatekeeper,C,reject(register),D),T_3):T_3<T_2+10\end{aligned}\tag{4.3}$$

SIC (4.1) says: if C asks *gatekeeper* to join the society (*register*), then the *gatekeeper* should ask for a registration $form$; SIC (4.2) imposes that, after the first two messages, the agent should provide the $form$; and SIC (4.3) says that, after receiving the form, the *gatekeeper* should either *accept* or *reject* the registration request.

[2] Time unit is an abstract concept, whose instantiation actually depends on the application. A time unit may represent for example a clock tick, or a transaction time.

For the sake of simplicity, in the sequel we assume that member agents do not leave the society. Then, the presence in the history of an event of type:

$$H(tell(gatekeeper, C, accept(register), D), T)$$

can be regarded as C's "formal" act of "membership", and it can be used in SICs as a condition for generating expectations.

For instance, SIC (3.15) from the FIPA-CNP (Sect. 3.1) could be modified as follows to take membership into account:

$$\begin{aligned}&\mathbf{H}(tell(gatekeeper, I, accept(register), D), T_I) \wedge \\ &\mathbf{H}(tell(I, P, cfp(R), D), T) \rightarrow \\ &\mathbf{E}(tell(P, I, propose(R, Q), D), T_1) : T_1 < T + 200 \vee \\ &\mathbf{E}(tell(P, I, refuse(R), D), T_2) : T_2 < T + 200\end{aligned} \tag{4.4}$$

5. Specification of the TCP protocol opening phase. In this section, we present a specification of the open-connection phase of the TCP protocol. We will focus on the well known "three-way handshake" opening, summarized below:

1. a peer A sends to another peer B a *syn* segment;[3]
2. B replies by acknowledging (with an *ack* segment) A's *syn* segment, and by sending a *syn* segment in turn;
3. A acknowledges B's *syn* segment with a *ack* segment, and starts sending data.

The following two integrity constraints describe such a protocol:

$$\begin{aligned}&\mathbf{H}(tell(A, B, tcp(syn, null, NSynA, AckNumber), D), T1) \rightarrow \\ &\mathbf{E}(tell(B, A, tcp(syn, ack, NSynB, NSynAAck), D), T2) : \\ &NSynAAck = NSynA + 1 \wedge T2 > T1.\end{aligned} \tag{5.1}$$

SIC 5.1 says that if A sends to B a *syn* segment, whose sequence number is $NSynA$, then B is expected to send to A an *ack* segment, whose acknowledgment number is $NSynA + 1$, at a later time.

$$\begin{aligned}&\mathbf{H}(tell(A, B, tcp(syn, null, NSynA, AckNumber), D), T1) \\ &\wedge \mathbf{H}(tell(B, A, tcp(syn, ack, NSynB, NSynAAck), D), T2) : \\ &T2 > T1 \wedge NSynAAck = NSynA + 1 \rightarrow \\ &\mathbf{E}(tell(A, B, tcp(null, ack, NSynAAck, NSynBAck), D), T3) : \\ &T3 > T2 \wedge NSynBAck = NSynB + 1.\end{aligned} \tag{5.2}$$

SIC 5.2 says that, if the previous two messages have been exchanged, then A is expected to send to B an *ack* segment acknowledging B's *syn* segment, and with acknowledgement number is $NSynB+1$, where $NSynB$ is the sequence number of B's *syn*.

A third integrity constraint has been added, to verify the interaction between peers with different response time. A faster peer in fact could not wait enough for the acknowledge message, and try to resend a syn message to a slower peer. This situation can lead to several problems in the slower peer, whose queue of the incoming messages could easily get saturated by requests.

$$\begin{aligned}&\mathbf{H}(tell(A, B, tcp(syn, null, NSynA, ANY), D), T1) \\ &\wedge ta(TA) \rightarrow \\ &\mathbf{EN}(tell(A, B, tcp(syn, null, NSynA, ANY), D), T2) : \\ &T2 < T1 \wedge T2 > T1 - TA.\end{aligned} \tag{5.3}$$

SIC 5.3 says that, if A has sent to B a *syn* segment to open a connection, then A is expected not to send another *syn* segment before TA time units, where TA is an application-specific constant, defined by the $ta/1$ predicate.

The above specification has been used to check the interaction between experimental mobile phones and a server.

[3] The term "segment" is used in the TCP specification to indicate bit configuration or streams.

TABLE 6.1
State of an expectation

Type	Verified	Expired	State
E	yes		fulfilled
E	no	no	wait
E	no	yes	violated
EN	yes		violated
EN	no	no	wait
EN	no	yes	fulfilled

6. Verification System. In this section, we describe a prototypical system that we have developed to verify the compliance of the agent behaviour to interaction protocols specified by means of SICs.

The system checks for compliance by accomplishing two main tasks:

1. it fires (*activates*) SICs whose conditions become true as relevant events occurs;
2. it decides whether *activated* SICs are fulfilled or violated.

The system is designed to work during the evolution of the society, so it will only have, at each instant, a partial history available, and it must take into account that new events may happen in the future. For instance, let us consider again the sample expectation in Sect. 2:

$$\mathbf{E}(accept(a_k, a_j, give(M), d_2), T_\mathrm{a}) : M \geq 10, T_\mathrm{a} \leq 15.$$

Let us now suppose that, at time 12, no matching event has yet occurred. So, while this expectation has not been fulfilled, neither it has (yet) been violated: since a matching event could still happen at time 13, 14 or 15. It will actually be violated instead, in case a matching event fails to occur by time 15, because the CLP constraint on the time variable becomes unsatisfiable as of time 16.

More generally, it may not be possible to state whether a SIC is fulfilled or violated at the same time it fires; thus, we identify three possible states for an activated SIC:

- *fulfilled*, if the SIC is fulfilled;
- *violated*, if the SIC is violated;
- *wait*, if the SIC is still neither fulfilled nor violated.

The initial state for an activated SIC is wait; happening events will eventually change its state to fulfilled or violated.

If we process events in the correct order in time, in the case of backward SICs, the transition from a wait state to a fulfilled or violated state is immediate, because expectations in a backward SIC regard events that should have (not) happened in the past and, thus, they can be immediately checked for fulfillment.

6.1. Runtime identification of the state of a SIC. In the following, we explain how the state of a SIC changes at runtime.

The activation of a SIC causes the creation of an instance of its "head" (organized in priority levels, each being a disjunction of conjunction of expectations, as explained in Sect. 2). Afterwards, the state of each single expectation is defined, followed by the state of the priority levels, and finally by the state of the SIC.

State of an expectation. An expectation is called "verified" if there exists a matching event in the society history. The state of a verified positive expectation is *fulfilled*; the state of a verified negative expectation is *violated*.

An expectation is called "expired" if CLP constraints over its time variable cannot be any longer satisfied (typically, this is the case with constraints representing deadlines which have expired). The state of an expired and not verified expectation is *violated* if the expectation is positive and *fulfilled* if the expectation is negative; the state of a not expired and not verified expectation is instead *wait*.

Table 6.1 summarises all these cases.

State of a conjunction of expectations. The state of a conjunction of expectations is defined by the following rules:

1. if the state of at least one expectation in the conjunction is *violated*, then the state of the conjunction is *violated*;
2. if the state of all expectations in the conjunction is *fulfilled*, the state of the conjunction is *fulfilled*;
3. otherwise, the state is *wait*.

State of a priority level. A priority level is a disjunction of conjunctions of expectations. The state of a priority level is then defined by the following rules:

1. if the state of at least one of the disjuncts is *fulfilled*, then the state of the priority level is *fulfilled*;
2. if the state of all of the disjuncts is *violated*, then the state of the priority level is *violated*;
3. otherwise, the state is *wait*.

State of a SIC. If all the priority levels of a SIC are violated, then the SIC is *violated*; otherwise, the state of the highest non-violated priority level of the SIC defines the state of the SIC.

6.2. Verification of Compliance. As shown in Sect. 3.1 in relation to the FIPA CNP, backward SICs can express that events are only allowed if some other events have (not) happened before; since their state can be immediately resolved to *fulfilled* or *violated*, backward SICs can be used to verify that an event is allowed as soon as it occurs. In designing our system, we made a choice to ignore the events that are not allowed. However, the system captures the violation: in a richer social model, we can imagine some authority to react to the violation.

The set of forward SICs associated with a legal action is then used to generate expectations about the future events in the society (i. e., the heads of associated forward SICs will be checked for fulfillment).

In order to verify the fulfillment of SICs, we have defined two different phases: the *Event Driven* phase and the *Clock Driven* phase.

Event-driven phase. An event-driven phase starts each time a new event occurs. The system activates all backward SICs associated with the event; if they are all fulfilled, then the event is recognized to be allowed and thus marked as "legal" and added to the history of the interaction. If some of the backward SICs are violated, then the event is marked as "illegal", since it is not allowed, and it is not recorded in the history of the society.

If the event is marked legal, the system processes the new updated history by activating the forward SICs associated with the new event. Forward (activated) SICs define the expected future behaviour of the society, and they will be checked for fulfillment.

Clock-driven phase. The clock-driven phase starts whenever a special event called "clock," or "current time," is registered by the society. The system processes the set of activated forward SICs identifying the state of each one. If the state of a SIC is fulfilled, the SIC is removed from the list of pending (waiting) SICs. If the state of a SIC is violated, the SIC is removed but a violation is raised. If the state is wait, the SIC is kept pending until the next clock-driven phase or the next event-driven phase. Note that the time associated to events and the "current time" event which fires a clock-driven phase must synchronize.

6.3. Implementation. The verification system has been implemented on top of SICStus Prolog's *Constraint Handling Rules* (*CHR*) library [22].

CHR[16] are essentially a committed-choice language consisting of guarded rules that rewrite constraints in a store into simpler ones until they are solved. *CHR* define both *simplification* (replacing constraints by simpler constraints while preserving logical equivalence) and *propagation* (adding new, logically redundant but computationally useful, constraints) over user-defined constraints.

6.3.1. Activation of SICs. Each event happened in the system is represented by the *CHR* constraint `h/2`, where the arguments are a Prolog ground term representing the happened event and an integer number representing the time.

Positive (resp. negative) expectations are represented by the Prolog term `e` (resp. `en`). Its arguments are: a Prolog term describing the event expected to happen (resp. not to happen), the time (typically non ground), and a list of CLP constraints over the variables in the description.

A *PriorityLevel* is represented by the Prolog term `pr`, whose arguments are the list of alternative *HeadDisjuncts* of the priority level and the integer number representing the priority (the lower the number, the higher the priority). Priority levels generated by a SIC are collected as the list argument of a `plist` term.

The argument of the *CHR* constraint `le/1` is the list of all activated `plists` (one for each activated SIC).

Each SIC is represented by a *simpagation CHR*. In general, simpagation rules have the form

$$H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \tag{6.1}$$

where $l > 0$, $i > l$, $j \geq 0$, $k \geq 0$ and where the multi-head $H_1, \ldots, H_i$ is a nonempty sequence of *CHR* constraints, the guard $G_1, \ldots, G_j$ is a sequence of built-in constraints, and the body $B_1, \ldots, B_k$ is a sequence of built-in and *CHR* constraints. Operationally, when the constraints in the head are in the constraint store and

the guard is true, $H_1, \ldots, H_l$ remain in the store, and $H_{l+1}, \ldots, H_i$ are substituted by $B_1, \ldots, B_k$. For instance, the following *CHR* implements SIC (2.2):

```
h(event0,T0), h(event1,T1) \ le(LExp) <=> T0<T1 &
append(LExp,
 [plist([
    pr([
      and([ e(event2,T2,[min(T2,T1)]) ]),
      and([ en(event3,T3,[min(T3,T0)]) ])
    ],1),
    pr([
      and([ e(event4,T4,[min(T4,T0)]) ])
    ],2)
  ],id1)], LExp1)
| le(LExp1).
```

If `event0` and `event1` have occurred and are part of the "history," the two *CHR* constraints `h(event0,T)` and `h(event1,T1)` are in the constraint store; if the guard `T<T1` is true, then the rule is activated. The store (the `LExp` list) of the heads of activated SICs is updated appending a new *plist()*, which contains the list of priority levels (two in this example) in the head of the SIC. The *CHR* constraint `le/1`, which contained the old `LExp` before the activation of the rule, is removed by simpagation and replaced by the same constraint with the new list `LExp1` as argument.

Note that two different symbols are used to represent the CLP constraint $<$: `<` if its arguments are the times of two happened events[4], and `min` if they are instead the times of two expectations.

The translation of a SIC into a simpagation *CHR* is rather straightforward, which makes it easy to implement new protocols.

As further examples, we report below the *CHR* implementation of SIC (3.1) and SIC (3.15):

```
h(tell(P,I,propose(R,Q),D),T) \
le(LExp) <=>
true &
append(LExp,
[plist([
     pr([
         and([
     e(tell(I,P,cfp(R),D),T1,[min(T1,T)])
             ])
        ],1)
])], LExp1) | le(LExp1).
```

```
h(tell(I,P,cfp(R),D),T) \
 le(LEv,LExp) <=>
 Td is T+200 &
 append(LExp,
 [plist([
     pr([
         and([
     e(tell(P,I,propose(R,Q),D),T1,[min(T1,Td)])
             ]),
         and([
     e(tell(P,I,refuse(R),D),T2,[min(T2,Td)])
             ])
        ],1)
   ])],
LExp1) | le(LExp1).
```

[4] In this case, the times are certainly ground and the Prolog predefined predicate can be applied to them.

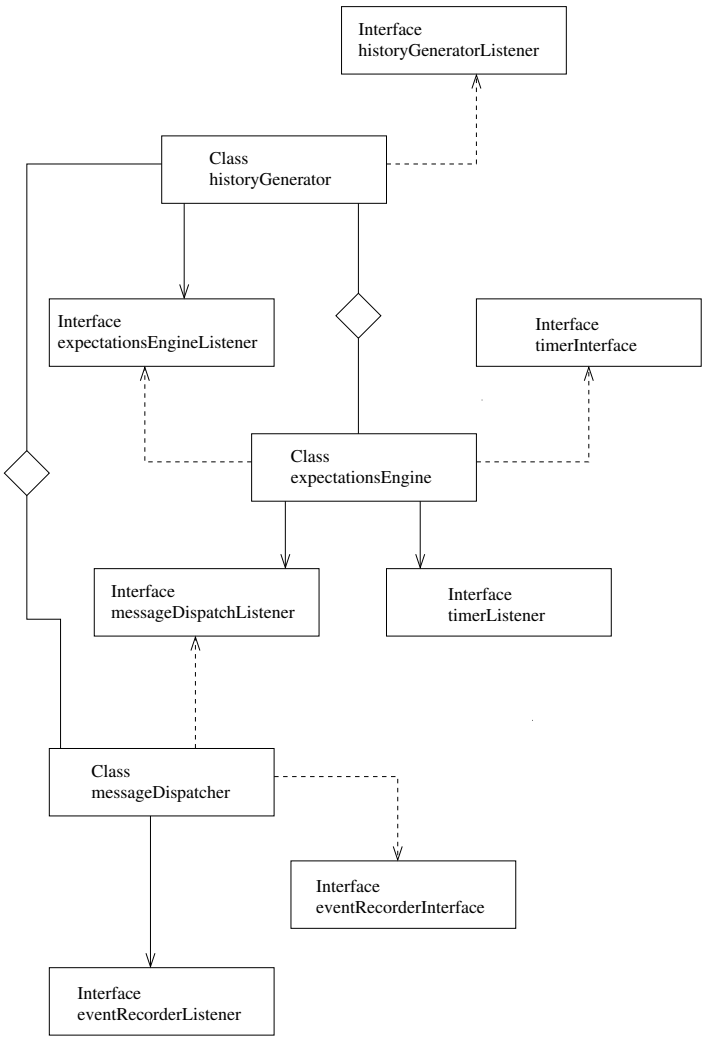

FIG. 6.1. *UML diagram*

6.3.2. Identification of the state of SICs. The identification of the state of a SIC is coded in standard Prolog. The system performs all the steps described in Sect. 6.1. It analyses all its stored `plists`, thus implementing the event-driven and clock-driven phases described above.

6.3.3. Interface to the verification system. In order to use the system in concrete case studies, a Java package (using the SICStus Prolog's Jasper library [22]) has been implemented. This package has been developed to be used as a Java wrapper for the verification system.

The UML diagram of the system is represented in Fig. 6.1. To use the system the user must create a *historyGenerator* object giving as parameter the path to a (compiled) Prolog file containing the protocol definition expressed by SICs. The Java system implements the Event Driven phase receiving messages from the *eventRecorderListener* interface and the clock-driven phase receiving "current time" events from the *timerListener* interface. The rest of the system implements the Java-Prolog interface.

7. Discussion and related work. The syntax of Social Integrity Constraints proposed in this paper is a modified version of that proposed in [2] and in [5]. The modifications have been made in order to tackle both expressiveness and implementation issues. Specifically:

- we added priority levels to SICs (see Sect. 2). This allows for a more flexible specification of protocols, enabling the protocol designer to devise alternative protocol flows while being able to specify preferences among them;

- we imposed the restriction of having only either backward or forward expectation in a SIC (see Sect. 2). While this improves efficiency, on the downside it prevents from writing SICs such as

$$\begin{aligned} &\mathbf{H}(a, T_a) \\ \rightarrow &\mathbf{E}(b, T_b) : T_b < T_a, 1 \\ \Rightarrow &\mathbf{E}(c, T_c) : T_c \leq T_a + \tau, 2 \end{aligned} \tag{7.1}$$

  which one might want to use to express that an event (b) that does not fulfill a backward expectation can, with lower priority, still be allowed, provided that certain "backup" event (c) occur at some point in the future. However, in our experience, SICs such as (7.1) are generally not necessary to express protocols of common use.

In [4] we have defined an abductive semantics for SICs, in the context of agent societies, and a more general framework, in which the verification procedure is performed by an abductive proof procedure [6], whose implementation has been integrated into a software component [3], interfaced to several multi-agent platforms such as Jade [8], PROSOCS [9], and tuProlog [12]. Other authors have proposed alternative approaches to the specification and in some cases animation of interaction among agents. Notably, in [7], Artikis et al. present a theoretical framework for providing executable specifications of particular kinds of multi-agent systems, called open computational societies, and they present a formal framework for specifying and animating 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. A noteworthy difference with [7] 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.

In [24], Yolum and Singh apply a variant of Event Calculus [19] 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, but 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 (which, in that work, is implemented by an abductive event calculus planner). Our notion of expectation is more general than that of commitment found in [24] or in other commitment-based works, such as [15]: it represents the necessity of a (past or future) event, and is not bound to have a debtor or a creditor, or to be brought about by an agent.

8. Conclusions. We have presented a framework for the specification and runtime verification of compliance of agent interaction to protocols. The specification at a social level of interaction protocols constrains the agent observable behaviour from the outside, rather than its internal state or structure. This is a characteristic of social approaches to agent protocol specification, and it is particularly suited for usage in open agent societies. Protocol specifications use a computational logic-based formalism called social integrity constraints. The system's Java-Prolog-*CHR*based implementation has been tested on different types of protocols [23]. In this article, we have demonstrated the usage of SICs in three cases: the FIPA CNP, taken from the agent literature, a made up protocol for joining semi-open societies, and the well known three-way handshake phase of the TCP IP protocol for connection establishment. The verification system, implemented in Prolog and *CHR*, can be used as a module in a Java-based system, thanks to the Java-Prolog interface of SICStus Prolog. The modular structure of the system makes it (hopefully) easy to adapt it to new applications.

9. Acknowledgments. This research has been partially supported by the National MIUR PRIN 2005 projects No 2005-011293, *Specification and verification of agent interaction protocols*, [5], and No 2005-015491, *Vincoli e preferenze come formalismo unificante per l'analisi di sistemi informatici e la soluzione di problemi reali*, [6] and by the National FIRB project *TOCAI.IT*[7].

[5] http://www.ricercaitaliana.it/prin/dettaglio_completo_prin_en-2005011293.htm
[6] http://www.sci.unich.it/~bista/projects/prin2006/
[7] http://www.dis.uniroma1.it/~tocai/

REFERENCES

[1] *FIPA Contract Net Interaction Protocol*, Tech. Report SC00029H, Foundation for Intelligent Physical Agents, 2002. Available at `http://www.fipa.org`

[2] 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. Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, V. Mařík, J. Müller, and M. Pěchouček, eds., vol. 2691 of Lecture Notes in Artificial Intelligence, Prague, Czech Republic, June 16–18 2003, Springer-Verlag, pp. 204–213.

[3] M. Alberti, M. Gavanelli, E. Lamma, F. Chesani, P. Mello, and P. Torroni, *Compliance verification of agent interaction: a logic-based software tool.*, Applied Artificial Intelligence, 20 (2006), pp. 133–157.

[4] 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, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa, A. Cappelli and F. Turini, eds., vol. 2829 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Sept. 23–26 2003, pp. 287–299.

[5] ———, *Specification and Verification of Agent Interactions using Social Integrity Constraints*, Electronic Notes in Theoretical Computer Science, 85 (2003).

[6] ———, *The $\mathcal{SCIFF}$ abductive proof-procedure*, in Proceedings of the 9th National Congress on Artificial Intelligence, AI*IA 2005, vol. 3673 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2005, pp. 135–147.

[7] A. Artikis, J. Pitt, and M. Sergot, *Animated specifications of computational societies*, in Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III, C. Castelfranchi and W. Lewis Johnson, eds., Bologna, Italy, July 15–19 2002, ACM Press, pp. 1053–1061.

[8] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, *Jade - a java agent development framework*, in Multi-Agent Programming: Languages, Platforms and Applications, R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, eds., vol. 15 of Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer-Verlag, 2005, pp. 125–147.

[9] A. Bracciali, U. Endriss, N. Demetriou, A. C. Kakas, W. Lu, and K. Stathis, *Crafting the mind of prosocs agents*, Applied Artificial Intelligence, 20 (2006), pp. 105–131.

[10] F. Chesani, A. Ciampolini, P. Mello, M. Montali, and S. Storari, *Testing guidelines conformance by translating a graphical language to computational logic*, in ECAI 2006 Workshop on AI techniques in healthcare:evidence based guidelines and protocols, Riva del Garda, Italy, August 2006. `http://www.openclinical.org/cgp2006_2.html`

[11] P. Davidsson, *Categories of artificial societies*, in Engineering Societies in the Agents World II, A. Omicini, P. Petta, and R. Tolksdorf, eds., vol. 2203 of Lecture Notes in Artificial Intelligence, Springer-Verlag, Dec. 2001, pp. 1–9. 2nd International Workshop (ESAW'01), Prague, Czech Republic, July 7, 2001, Revised Papers.

[12] E. Denti, A. Omicini, and A. Ricci, *Multi-paradigm Java-Prolog integration in* tu*Prolog*, Science of Computer Programming, 57 (2005), pp. 217–250.

[13] U. Endriss, N. Maudet, F. Sadri, and F. Toni, *Protocol conformance for logic-based agents*, in Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico (IJCAI-03), G. Gottlob and T. Walsh, eds., Morgan Kaufmann Publishers, Aug. 2003.

[14] *FIPA: Foundation for Intelligent Physical Agents.* Home Page: `http://www.fipa.org/`

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

[16] T. Frühwirth, *Theory and practice of constraint handling rules*, Journal of Logic Programming, 37 (1998), pp. 95–138.

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

[18] A. C. Kakas, R. A. Kowalski, and F. Toni, *The role of abduction in logic programming*, in Handbook of Logic in Artificial Intelligence and Logic Programming, D. M. Gabbay, C. J. Hogger, and J. A. Robinson, eds., vol. 5, Oxford University Press, 1998, pp. 235–324.

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

[20] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 2nd extended ed., 1987.

[21] J. Muller and J. Odell, *Agent UML: A formalism for specifying multiagent software systems*, International Journal of Software Engineering and Knowledge Engineering, 11(3) (2001), pp. 207–230.

[22] *SICStus prolog user manual, release 3.11.0*, Oct. 2003. `http://www.sics.se/isl/sicstus/`

[23] *The SOCS protocol repository*, 2005. Available at `http://edu59.deis.unibo.it:8079/SOCSProtocolsRepository/jsp/index.jsp`

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

Edited by: Marcin Paprzycki, Niranjan Suri
Received: October 1, 2006
Accepted: December 10, 2006