Specification and Verification of Agent Interaction Protocols in a Logic-based System

Marco Alberti
DI,Università di Ferrara
Via Saragat,1
44100 Ferrara (Italy)
malberti@ing.unife.it

Davide Daolio
DI,Università di Ferrara
Via Saragat,1
44100 Ferrara (Italy)

Paolo Torroni
DEIS, Università di Bologna
Via del Risorgimento, 2
40136, Bologna (Italy)
ptorroni@deis.unibo.it

ABSTRACT

In multiagent systems, agent interaction is ruled by means of interaction protocols. Compliance to protocols can be hardwired in agent programs; however, this requires that only "certified" agents interact. In open societies, composed of autonomous and heterogeneous agents whose internal structure is, in general, not accessible, interaction protocols should be specified in terms of the agent *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 agents' behaviour to protocols specified in a logic-based formalism (*Social Integrity Constraints*). We also present the application of the formalism and the system to the specification and verification of the FIPA Contract-Net protocol.

Keywords

Agents, Verification, Computational Logic

1. INTRODUCTION

In multiagent systems, agent interaction is usually subject to some kind of interaction protocols, which agents should respect when interacting. This raises the obvious problem of verifying that agents actually respect interaction protocols.

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 [6], Endriss et al. propose an approach where protocols are "imported" into individual agents' 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 [4, 3], we proposed a computational logic-based formalism (based upon *Social Integrity Constraints*) 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 Social Integrity Constraints, as long as its behaviour is compliant.

For the purposes of this work, we modified Social Integrity Constraints, in order to tackle the expressive needs of practical applications and to allow a more efficient implementation of compliance verification procedures.

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 propose an implementation of the FIPA Contract Net protocol [1] by means of Social Integrity Constraints.

We then present, in Sect. 4, a system for the verification of compliance to Social Integrity Constraints that has been implemented by using SICStus Prolog [2] and its *Constraint Handling Rules* [9] library.

Discussion of related work and conclusions follow.

2. SOCIAL INTEGRITY CONSTRAINTS

In this section, we explain the concepts that we use to represent an agent's actual behaviour (*events*) and desired behaviour (*expectations*); we then explain how Social Integrity Constraints (also SICs, for short, in the following) express which expectations are generated as consequence of events.

Events and expectations. In our framework, the agent behaviour is represented by means of *events*.

Events are of the form $\mathbf{H}(Description, Time)$, where *Description* is a term (as intended in logic programming, see [12]) representing the event that has happened, and *Time* is an integer number representing the time at which the event has happened.

For example, $\mathbf{H}(request(a_i, a_j, give(10\$), d_1), 7)$ represents the fact that agent a_i requested agent a_j to give 10$, in the

[1]We intend *openness* in societies of agents as Artikis, Pitt and Sergot [5], where agents can be heterogeneous and possibly non-cooperative.

SAC'04, March 14-17, 2004, Nicosia, Cyprus

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 $\mathbf{E}(Description, Time)$ 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 $\mathbf{EN}(Description, Time)$ and represent an event that 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 [10] constraints can be imposed on variables to restrict their domain.

For instance,

$$\mathbf{E}(accept(a_k, a_j, give(M), d_2), T_a) : M \geq 10, T_a \leq 15 \quad (1)$$

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 (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 lesser or equal than 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, it should be noticed here that expectations regard only point-time events; thus it is not possible, in our framework, to express explicitly that some *proposition* is expected to be true in a given time interval, as in other frameworks [11, 8].

Since we make no assumptions about agents' internal structure or policies, their behaviour may satisfy expectations, or not. 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);
- CLP constraints on variables (if any) are satisfied.

A positive expectation matched by an event is *fulfilled*; a negative expectation matched by an event is *violated*.

For instance, event $\mathbf{H}(accept(a_k, a_j, give(20), d_2), 15)$ fulfills expectation (1); the same event would, instead, violate a negative expectations with the same content and CLP constraints.

At the end of the history of a society (i.e., when it is assumed that no more events can happen), all positive expectations that are not fulfilled are violated, and all negative expectations that are not violated are fulfilled.

Social Integrity Constraints. The way expectations should be generated, given a partial history of a society, is specified by means of *Social Integrity Constraints*. SICs, as presented here, are a modified version of those introduced in [3]: we discuss and motivate the modifications here introduced in Sect. 5.

In Table 1, the BNF syntax of SICs is given. **Term** is a term as intended in logic programming [12], **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

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)

Table 1: BNF syntax of Social Integrity Constraints

of expectations (*HeadDisjuncts* in Table 1) are alternative, and to assign a priority, represented by an integer number, to each list of alternatives (*PriorityLevel*s in Table 1).

For instance, the following SIC:

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

means that, if e_0 happens before e_1:

- preferably, e_2 should have happened before e_1 or e_3 should not have happened before e_0,
- or, with lower priority, e_4 should have happened before e_0.

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 priority, the better). In this case, we say that the SIC is *fulfilled*; otherwise, it is *violated*. While priorities have no effect upon the declarative semantics of SICS (with respect to fulfillment or violation), they can be used by the compuational entity representing the society to guide the agent behaviour towards a preferred state. For instance, a society which can communicate expectations to agents can choose a set of expectations which, if fulfilled, satisfies the highest priority level. This aspect is not addressed in this work.

The expectations in SIC (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. 5.

3. EXAMPLE: FIPA CONTRACT-NET

In this section, we present an implementation of FIPA Contract Net protocol [1] by means of SICs.

3.1 FIPA-CN protocol flow

FIPA-CN is a protocol based on FIPA-ACL [7] defined for regulating transactions between entities using negotiation. The protocol flow, represented as an AUML [13] diagram in Fig. 1, starts with an Initiator which issues a request for a resource (*cfp*[2]) to other Participants. The Participants can reply proposing a price for satisfying the re-

[2] *cfp* stands for Call For Proposals.

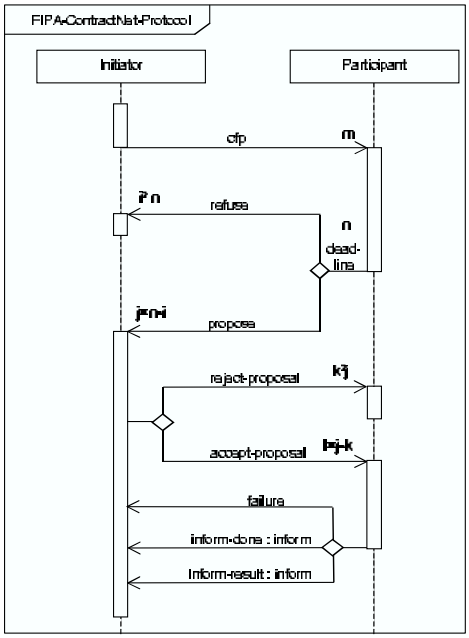

Figure 1: FIPA-Contract-Net Interaction Protocol (AUML Diagram)

quest (*propose*), or refusing (*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 of having provided for the resource (with an *inform-done* or a more informative *inform-result*) or of having failed to provide for the resource (*failure*).

3.2 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. The choice of SICs seems satisfactory, although it is not the only possible one. We are currently investigating a general mapping of AUML protocol diagrams to SICs, so as to allow for an automatic translation.

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 (possibly with subscripts) the time. The integer number representing the priority is not necessary because each SIC only has one priority level, and has thus been omitted for ease of reading.

Backward SICs. Backward SICs are used to express that an action is only allowed if some other actions have (or have not) been performed before.

SICs (3) and (4) state that *propose* and *refuse* are only allowed in reply to a *cfp*.

$$\mathbf{H}(tell(P, I, propose(R, Q), D), T) \rightarrow \\ \mathbf{E}(tell(I, P, cfp(R), D), T_1) : T_1 < T \tag{3}$$

$$\mathbf{H}(tell(P, I, refuse(R), D), T) \rightarrow \\ \mathbf{E}(tell(I, P, cfp(R), D), T_1) : T_1 < T \tag{4}$$

SICs (5) and (6) express mutual exclusiveness between *propose* and *refuse*.

$$\mathbf{H}(tell(P, I, propose(R, Q), D), T) \rightarrow \\ \mathbf{EN}(tell(P, I, refuse(R), D), T_1) : T_1 \leq T \tag{5}$$

$$\mathbf{H}(tell(P, I, refuse(R), D), T) \rightarrow \\ \mathbf{EN}(tell(P, I, propose(R, Q), D), T_1) : T_1 \leq T \tag{6}$$

SICs (7) and (8) state that *accept-proposal* and *reject-proposal* are only allowed in reply to a *propose*.

$$\mathbf{H}(tell(I, P, accept\text{-}proposal(R, Q), D), T) \rightarrow \\ \mathbf{E}(tell(P, I, propose(R, Q), D), T_1) : T_1 < T \tag{7}$$

$$\mathbf{H}(tell(I, P, reject\text{-}proposal(R, Q), D), T) \rightarrow \\ \mathbf{E}(tell(P, I, propose(R, Q), D), T_1) : T_1 < T \tag{8}$$

SICs (9) and (10) express mutual exclusiveness between *accept-proposal* and *reject-proposal*.

$$\mathbf{H}(tell(I, P, accept\text{-}proposal(R, Q), D), T) \rightarrow \\ \mathbf{EN}(tell(I, P, reject\text{-}proposal(R, Q), D), T_1) : T_1 \leq T \tag{9}$$

$$\mathbf{H}(tell(I, P, reject\text{-}proposal(R, Q), D), T) \rightarrow \\ \mathbf{EN}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 \leq T \tag{10}$$

SICs (11), (12) and (13) say that *inform-done*, *inform-result* and *failure* are only allowed in reply to an *accept-proposal*.

$$\mathbf{H}(tell(P, I, inform\text{-}done(R), D), T) \rightarrow \\ \mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T \tag{11}$$

$$\mathbf{H}(tell(P, I, inform\text{-}result(R, S), D), T) \rightarrow \\ \mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T \tag{12}$$

$$\mathbf{H}(tell(P, I, failure(R), D), T) \rightarrow \\ \mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T \tag{13}$$

SICs (14), (15) and (16) express mutual exclusiveness between *inform-done*, *inform-result* and *failure*.

$$\mathbf{H}(tell(P, I, inform\text{-}done(R), D), T) \rightarrow \\ \mathbf{EN}(tell(P, I, failure(R), D), T_1) : T_1 \leq T \wedge \\ \mathbf{EN}(tell(P, I, inform\text{-}result(R, S), D), T_1) : T_1 \leq T \tag{14}$$

$$\mathbf{H}(tell(P, I, inform\text{-}result(R, S), D), T) \rightarrow \\ \mathbf{EN}(tell(P, I, failure(R), D), T_1) : T_1 \leq T \wedge \\ \mathbf{EN}(tell(P, I, inform\text{-}done(R), D), T_1) : T_1 \leq T \tag{15}$$

$$\mathbf{H}(tell(P, I, failure(R), D), T) \rightarrow \\ \mathbf{EN}(tell(P, I, inform\text{-}done(R), D), T_1) : T_1 \leq T \wedge \\ \mathbf{EN}(tell(P, I, inform\text{-}result(R, S), D), T_1) : T_1 \leq T \tag{16}$$

Forward SICs. SIC (17) says that, after receiving a *cfp*, a Participant is expected to issue a *propose* or a *refuse* by 200 clock ticks.

$$\mathbf{H}(tell(I, P, cfp(R), D), T) \rightarrow \\ \mathbf{E}(tell(P, I, propose(R, Q), D), T_1) : T_1 < T + 200 \vee \\ \mathbf{E}(tell(P, I, refuse(R), D), T_2) : T_2 < T + 200 \tag{17}$$

SIC (18) states that the Initiator is expected to reply to a *propose* with an *accept-proposal* or a *reject-proposal* by 200 clock ticks.

$$\mathbf{H}(tell(P, I, propose(R, Q), D), T) \rightarrow$$
$$\mathbf{E}(tell(I, P, accept\text{-}proposal(R, Q), D), T_1) : T_1 < T + 200 \vee$$
$$\mathbf{E}(tell(I, P, reject\text{-}proposal(R, Q), D), T_2) : T_2 < T + 200 \quad (18)$$

SIC (19) 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.

$$\mathbf{H}(tell(I, P, accept\text{-}proposal(R, Q), D), T) \rightarrow$$
$$\mathbf{E}(tell(P, I, inform\text{-}done(R), D), T_1) : T_1 < T + 200 \vee$$
$$\mathbf{E}(tell(P, I, inform\text{-}result(R, S), D), T_2) : T_2 < T + 200 \vee$$
$$\mathbf{E}(tell(P, I, failure(R), D), T_2) : T_2 < T + 200 \quad (19)$$

It can be noticed that, in all the three cases, backward SICs make the alternative expectations mutually exclusive.

4. VERIFICATION SYSTEM

In this section, we describe a prototypical system that has been 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. *Activates* a SIC when the history of the society contains events that make its condition true;
2. Decide 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, that we report here for ease of reading:

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

Let us suppose that, at time 12, no event fulfilling it has happened. Yet, it is not correct to state that the expectation is violated, because an event fulfilling it could still happen at time 13, 14 or 15. Instead, at time 16, if no event fulfilling the expectation has happened, then the expectation is actually violated, because the CLP constraint on the time variable has become unsatisfiable.

More generally, it may not be possible to state whether a SIC is fulfilled or violated as soon as it is activated; 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.

For 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 (or have not) happened in the past and, thus, they can be immediately checked for fulfillment.

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

Table 2: State of an expectation

4.1 Runtime identification of the state of a SIC

In the following, we explain how the state of a SIC is determined during the computation.

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, it is defined the state of each single expectation, then the state of the priority levels, and finally the state of the SIC.

State of an expectation. An expectation is verified if there exists, in the history of the society, an event matching it. The state of a verified positive expectation is *fulfilled*; the state of a verified negative expectation is *violated* (see Sect. 2).

An expectation is expired if CLP constraints over its time variable cannot be satisfied (typically, if these constraints represent an expired deadline). The state of an expectation which is expired and not verified is *violated* if the expectation is positive and *fulfilled* if the expectation is negative; the state of an expectation which is not expired and not verified is *wait*.

Table 2 summarises all these cases.

State of a conjunction of expectations. A priority level is a disjunction of conjunctions of expectations.

The state of a conjunction of expectations is defined by the first of the following rules that applies:

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. The state of a priority level is defined by the first of the following rules that applies:

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 (*fulfilled* or *wait*) of the highest non-violated priority level of the SIC defines the state of the SIC.

4.2 Verification of Compliance

As shown in Sect. 3.2 for the FIPA Contract-Net protocol, backward SICs can express that actions are only allowed if some events have (or 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 happens.

In the design of our system, the choice has been made to deactivate (i.e., to not consider in the history) 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. The Event Driven phase is activated each time a new event happens.

The system activates all backward SICs associated with the event; if all of these are fulfilled, then the event is 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", and it is not recorded in the history of the society.

If the event is legal, the system processes the new updated history 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 is activated at each clock tick (i.e., whenever the current time in the society is updated). 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 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 as pending until next Clock Driven phase. It is important to notice that the time associated to events and the clock-time given in the Clock Driven phase must be coherent: in this work, we assume that time is unique inside a society, and that it is kept in a centralized timer.

4.3 Implementation

The verification system has been implemented by using SICStus Prolog [2], and, in particular, its *Constraint Handling Rules* library.

Constraint Handling Rules [9] (*CHR* for brevity hereafter) 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.

The main intended use for *CHR* is to write constraint solvers, or to extend existing ones. However, although ours is not a classic constraint programming setting, the computational model of *CHR* presents features that make it a useful tool for the verification of compliance to SICs.

4.3.1 Activation of SICs.

Constraint Handling Rules have been used to realize the activation system of backward and forward 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* (see Table 1) 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

$$H_1, \ldots, H_l \backslash H_{l+1}, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j | B_1, \ldots, B_k \quad (20)$$

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* represents sample SIC (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 happened, the two *CHR* constraints `h(event0,T)` and `h(event1,T1)` will be present 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.

It should be noticed that two different symbols are used to represent the CLP constraint $<$: `<` if the constraint is between the times of two happened events[3], and `min` if it is

[3]In this case, the times are certainly ground and the Prolog

between the times of two expectations.

It can be noticed that the translation of a SIC into a simpagation *CHR* is rather straightforward, thus making it easy to implement new protocols.

As further examples, we report the translations into *CHR*s of backward SIC (3) and forward SIC (17):

```
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).
```

4.3.2 *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. 4.1. The system analyses all the `plist`s stored in the system according to the definition and behaviour of Event Driven phase and Clock Driven phase (see Sect. 4.2).

4.3.3 *Interface to the verification system.*

In order to use the system in concrete case studies, a Java package (using the Jasper library of SICStus Prolog [2]) 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. 2. To use the system the user must create an object *historyGenerator* giving as parameter a (compiled) Prolog file containing the protocol definition expressed by SICs. The Java system implements the Event Driven phase receiving messages from the interface *eventRecorderListener* and the Clock Driven phase receiving platform clock from *timerListener* interface. The rest of the system implements the Java-Prolog interface.

5. DISCUSSION AND RELATED WORK

The syntax of Social Integrity Constraints proposed in this paper is a modified version of that proposed in [3] and in [4]. The modifications have been made in order to tackle both expressiveness and implementation issues. Specifically:

predefined predicate can be applied to them.

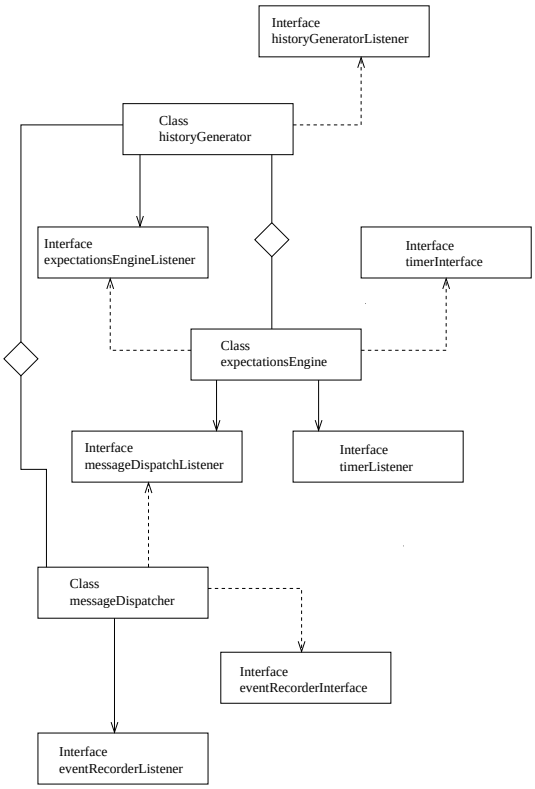

Figure 2: UML diagram

- 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 a preferred one;
- we imposed the restriction of having only either backward or forward expectation in a SIC (see Sect. 2): this prevents the protocol designer from writing such SICs as

$$\begin{aligned} &\mathbf{H}(a, T_a) \\ \rightarrow &\mathbf{E}(b, T_b) : T_b < T_a, 1 \\ \Rightarrow &\mathbf{E}(c, T_c) : T_c \leq T_a + \tau, 2 \end{aligned} \tag{21}$$

  which might be useful to express that an event that does not fulfill a backward expectation can, with lower priority, be "recovered" in the future. However, in the practical cases that we have considered so far, SICs as (21) have not been necessary.

In [5], 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 [5] 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 [14], Yolum and Singh apply a variant of Event Calculus [11] 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 [14] or in other commitment-based works, such as [8]: 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.

6. CONCLUSIONS

In this paper, we presented a Java-Prolog-*CHR* based system for the online verification of compliance of agent interaction to protocols.

The implemented system has been tested on different types of protocols: FIPA Contract-Net (described in this paper), the Dining Philosophers Problem and connection establishment in the TCP protocol.

The specification at a social level of interaction protocols constrains the agent observable behaviour rather than their internal state or structure, thus allowing for interaction in open societies. The specification is grounded on computational logic-based Social Integrity Constraints.

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.

In future work, we intend to address the notification of expectations to agents, so as to let the computational entity representing the society guide agent behaviour to a preferred state (as expressed by priorities).

7. ACKNOWLEDGMENTS

This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32530 project.

We thank the anonymous reviewers for their useful comments.

8. ADDITIONAL AUTHORS

Additional authors: Marco Gavanelli (DI, Università di Ferrara, email: `mgavanelli@ing.unife.it`), Evelina Lamma (DI, Università di Ferrara, email: `elamma@ing.unife.it`), Paola Mello (DEIS, Università di Bologna, email: `pmello@deis.unibo.it`)

9. REFERENCES

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

[2] SICStus Prolog 3.9.0 Manual. February 2002.

[3] M. Alberti, A. Ciampolini, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. A social acl semantics by deontic constraints. In V.Marik, J.Muller, and M.Pechoucek, editors, *Proceedings of the 3rd International/Central and Eastern European Conference on Multi-Agent Systems*, number 2691 in Lecture Notes in Artificial Intelligence, pages 204–213, Prague, Czech Republic, June 2003. Springer Verlag.

[4] M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. Specification and verification of agent interactions using social integrity constraints. In W. van der Hoek, A. Lomuscio, E. de Vink, and M. Wooldridge, editors, *Proceedings of the Workshop on Logic and Communication in Multi-Agent Systems (LCMAS)*, Eindhoven, the Netherlands, June 29 2003.

[5] A. Artikis, J. Pitt, and M. Sergot. Animated specifications of computational societies. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 1053–1061, Bologna, Italy, 2002. ACM.

[6] U. Endriss, N. Maudet, F. Sadri, and F. Toni. Protocol conformance for logic-based agents. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pages 679–684. Morgan Kaufmann Publishers, August 2003.

[7] FIPA Communicative Act Library Specification. Experimental specification XC00037H, Foundation for Intelligent Physical Agents, Aug. 2001. Published on August 10th, 2001, available for download from the FIPA website: `http://www.fipa.org`.

[8] 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 Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 535–542, Bologna, Italy, July 15–19 2002.

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

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

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

[12] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 2nd extended edition, 1987.

[13] J. Muller and J. Odell. Agent UML: A formalism for specifying multiagent interaction. *Agent-Oriented Software Engineering*, pages 91–103, 2001.

[14] P. Yolum and M. Singh. Flexible protocol specification and execution: applying event calculus planning using commitments. In C. Castelfranchi and W. Lewis Johnson, editors, *Proceedings of the First Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 527–534, Bologna, Italy, 2002.