Attribute Global Types for Dynamic Checking of Protocols in Logic-based Multiagent Systems

Viviana Mascardi and Davide Ancona
DIBRIS, Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
Via Dodecaneso, 35, 16146 Genova - ITALY
(*e-mail:* `viviana.mascardi@unige.it, davide.ancona@unige.it`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

This paper introduces Attribute Global Types, an extension inspired by Attribute Grammars to a formalism we have recently proposed for specifying and dynamically verifying multi-party agent interaction protocols. Global types equipped with attributes are more expressive, since they allow parametric specifications of protocols, but despite their expressive power, they can be still effectively used for dynamic checking of protocols: Attribute Global Types can be easily represented as Prolog terms, and a mechanism for verifying that a sequence of messages complies to an Attribute Global Type has been designed and implemented in Prolog. This logic-based representation and implementation allow us to integrate a monitor agent implementing a run-time verification mechanism of protocol compliance into any logic-based agent oriented programming language that supports the basic Prolog built-ins.

KEYWORDS: global types, protocol verification, logic-based multiagent system

1 Introduction

Multiagent systems (MASs, (Jennings et al. 1998)) are an industrial-strength technology for integrating and coordinating autonomous and heterogeneous systems. As agents are expected to be able to reason on what is happening in their surrounding environment and inside themselves, a logic-based approach to their specification and implementation has been often followed (Kowalski and Sadri 1999). For the purposes of this paper, we define a logic-based agent as an agent implemented in a computational logic-based language.

Since MASs are open, highly dynamic, and unpredictable, ensuring conformance of the agents' actual behavior to a given interaction protocol is of paramount importance to guarantee the participants' interoperability and security. Global types (Carbone et al. 2007; Honda et al. 2008; Castagna et al. 2012) are a behavioral type and process algebra approach to the problem of specifying and verifying multiparty interactions between distributed components. In (Ancona et al. 2012) we tackled the problem of run-time verification of the conformance of a MAS implementation to a specified protocol by exploiting basic global types on top of the Jason agent oriented logic programming language (Bordini et al. 2007). An extension of the formalism with sending action types and constrained shuffle was presented in (Ancona et al. 2013a) and refined in (Ancona et al. 2013b).

In this paper we further extend our formalism by adding attributes to both sending action types and constrained global types. Our proposal is mainly inspired by attribute grammars (Knuth 1990), a formal way to define attributes for the productions of a formal grammar, associating these attributes with values. As significant examples, we provide compact specifications parametric in the number of participants for non trivial protocols like the Alternating Bit Protocol (ABP) and the FIPA Iterated Contract Net Protocol (FIPA 2002b). The main strengths of our formalism are, in fact, the conciseness and readability, and the expressiveness of specifications, as already manifested in (Ancona et al. 2013b), with a comparison with other formalisms based on global types. With respect to the formalisms for representing protocols in MASs discussed in Section 5, to the best of our understanding none of them supports a synchronization operator with the same expressive power of our constrained shuffle, which is necessary to concisely describe complex protocols as the ABP without an exponential growth of the specification in the number of different kinds of exchanged messages.

Any protocol in our formalism can be expressed as a set of equations involving variables and first-order logic terms and the verification that actual messages exchanged among agents comply to the protocol has been encoded in Prolog by means of 15 lines of code faithfully implementing the attribute global types transition rules. Implementing a logic-based agent able to monitor the actual conversation among logic-based agents in the MAS is rather straightforward as well, as it basically amounts to solve the problem[1] of exhaustively inspecting the exchanged messages. This is why we think of logic-based MASs as the primary target of our work.

The paper is organized in the following way: Section 2 introduces our formalism, Section 3 describes its extension with attributes and provides examples. Section 4 discusses how attribute global types can be represented and verified in a Prolog-like language. Finally, Section 5 surveys the related approaches and draws the directions for future research.

2 Background: Constrained Global Types

In this section we present the formalism of constrained global types introduced by Ancona, Mascardi and Barbieri (Ancona et al. 2013b). Intuitively, a constrained global type represents the state of an interaction protocol from which several transition steps to other states (that is, to other constrained global types) are possible, with a resulting sending action. The syntax is based on the following building blocks.

Sending actions. A sending action a is a communicative event taking place between two agents and consisting of the sender and the receiver of the message, the performative expressed in some agent communication language such as FIPA-ACL (FIPA 2002a) or KQML (Mayfield et al. 1995), and the actual content of the message expressed in some content language shared among the agents, such as Prolog.

Sending action types. Sending actions types model the message pattern expected at a certain point of the conversation. A sending action type α is a predicate on sending actions. Its interpretation is the set of sending actions that verify α; we write $a \in \alpha$ to mean that α is true on a, and we also say that a has type α.

[1] This is always required when specifications are checked at runtime through system monitoring.

Producers and consumers. In order to model constraints across different branches of a constrained fork (explained later in this section), we introduce two different kinds of sending action types, called *producers* and *consumers*, respectively. Each occurrence of a producer sending action type must correspond to the occurrence of a new sending action; in contrast, consumer sending action types correspond to the same sending action specified by a certain producer sending action type. The purpose of consumer sending action types is to impose constraints on sending action sequences, *without introducing new events.* A consumer is a sending action type α, whereas a producer is a sending action type α equipped with a natural superscript n.

Constrained global types. A constrained global type τ represents a set of possibly infinite sequences of sending actions, and is defined on top of the following type constructors:

- λ (empty sequence): the singleton set $\{\epsilon\}$ containing the empty sequence ϵ.
- $\alpha^n{:}\tau$ (*seq-prod*): the set of all sequences whose first element is a sending action a matching type α ($a \in \alpha$), and the remaining part is a sequence in the set represented by τ. The superscript n specifies the number n of corresponding consumers that coincide with the same sending action type α; hence, n is the least required number of times $a \in \alpha$ has to be "consumed" to allow a transition labeled by a.
- $\alpha{:}\tau$ (*seq-cons*): a consumer of sending action a matching type α ($a \in \alpha$), and followed by any sequence in the set represented by τ.
- $\tau_1 + \tau_2$ (*choice*): the union of the sequences of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*fork*): the set obtained by shuffling the sequences in τ_1 with those in τ_2.
- $\tau_1 \cdot \tau_2$ (*concat*): the concatenation of the sequences of τ_1 and τ_2.

Constrained global types are regular terms, that is, can be cyclic (recursive), and hence they can be represented by a finite set of syntactic equations, as happens in most modern Prolog implementations. To make the treatment simpler, we limit our investigation to *contractive* (a.k.a. *guarded*) and *deterministic* types. A constrained global type τ is *contractive* if all infinite paths[2] in τ contain an occurrence of the : constructor. Determinism ensures that dynamic checking can be performed efficiently without backtracking. Intuitively, a constrained global type is deterministic if, in case more transition rules can be applied when sending action a takes place, they lead to equivalent global types. The formal definition is given in the next section.

Semantics. Type interpretation is based on the notion of transition, a total function $\delta{:}\mathbb{N} \times \mathcal{CT} \times \mathcal{A} \rightarrow \mathcal{P}_{fin}(\mathcal{CT} \times \mathbb{N})$, where $\mathcal{CT}$ and $\mathcal{A}$ denote the set of contractive and constrained global types and of sending actions, respectively. Figure 1 defines the rules for δ. If τ_1 represents the current state of the protocol and the sending action a takes place, then the protocol can move to τ_2 iff $\delta(0, \tau_1, a) = (\tau_2, 0)$, that we write as $\tau_1 \overset{a}{\rightarrowtail} \tau_2$.

The auxiliary function $\epsilon(\_)$, inductively defined in Figure 2, specifies the global types whose interpretation contains the empty sequence ϵ. Intuitively, a global type τ s.t. $\epsilon(\tau)$ holds specifies a protocol that is allowed to successfully terminate.

Let τ_0 be a contractive and constrained global type. A *run* ρ for τ_0 is a sequence $\tau_0 \overset{a_0}{\rightarrowtail} \tau_1 \overset{a_1}{\rightarrowtail} \ldots \overset{a_{n-1}}{\rightarrowtail} \tau_n \overset{a_n}{\rightarrowtail} \tau_{n+1} \overset{a_{n+1}}{\rightarrowtail} \ldots$ such that (1) either the sequence is infinite, or it

[2] By "path of a global type" we mean "path in the possibly infinite tree corresponding to the term".

has finite length $k \geq 0$ and the last global type τ_k verifies $\epsilon(\tau_k)$; and (2) for all τ_i, a_i, and τ_{i+1} in the sequence, $\tau_i \overset{a_i}{\rightarrowtail} \tau_{i+1}$ holds. We denote by $A(\rho)$ the possibly empty or infinite sequence of sending actions $a_0 a_1 \ldots a_n \ldots$ contained in ρ. The interpretation $[\![\tau_0]\!]$ of τ_0 is the set $\{A(\rho) \mid \rho$ is a run for τ_0 $\}$. A contractive global type τ is deterministic if for any possible run ρ of τ and any possible τ' in ρ, if $\tau' \xrightarrow{a} \tau''$, and $\tau' \xrightarrow{a} \tau'''$, then $[\![\tau'']\!] = [\![\tau''']\!]$.

Examples. The extension of "plain" global types introduced in (Ancona et al. 2012) with a constrained fork, was motivated by interaction protocols like the ABP. In ABP$_2$, four different sending actions may occur: Alice sends msg1 to Bob (sending action type *msg$_1$*), Alice sends msg2 to Bob (sending action type *msg$_2$*), Bob sends ack1 to Alice (sending action type *ack$_1$*), Bob sends ack2 to Alice (sending action type *ack$_2$*). The ABP is an infinite iteration, where the following constraints have to be satisfied for all occurrences of the sending actions:

- The n-th occurrence of *msg$_1$* must precede the n-th occurrence of *msg$_2$*.
- The n-th occurrence of *msg$_1$* must precede the n-th occurrence of *ack$_1$*, which, in turn, must precede the $(n+1)$-th occurrence of *msg$_1$*.
- The n-th occurrence of *msg$_2$* must precede the n-th occurrence of *ack$_2$*, which, in turn, must precede the $(n+1)$-th occurrence of *msg$_2$*.

Representing the ABP$_2$ without a constrained fork requires to take all the combinations of sending actions into account in an explicit way. The size of the type grows exponentially with the number of the different sending action types involved in the protocol.

With constrained fork the ABP$_2$ can be specified in a very compact and readable way: each of the three constraints above are combined together with a constrained fork, where producers and consumers are used to indicate the sending action types that have to correspond to the same event. The result for the ABP$_2$ with four messages is the following:

$$\begin{array}{ll} ABP_2 = MA_1|MA_2|MM & MA_2 = msg_2^1{:}ack_2^0{:}MA_2 \\ MA_1 = msg_1^1{:}ack_1^0{:}MA_1 & MM_2 = msg_1{:}msg_2{:}MM_2 \end{array}$$

msg_1^1 and msg_1 in MA_1 and MM respectively, always correspond to the same event (and analogously for msg_2^1 and msg_2 in MA_2 and MM). Since no constraint relates ack_1 and ack_2, the corresponding producers in MA_1 and MA_2 are super-scripted by 0.

ABP$_2$ can be easily generalized to ABP$_k$ where $2k$ different sending action types (with $k \geq 2$) are exchanged:

$$\begin{array}{c} ABP_k = (MA_1|\ldots|MA_k)|MM_k \qquad MM_k = msg_1{:}\ldots{:}msg_k{:}MM_k \\ MA_i = msg_i^1{:}ack_i^0{:}MA_i \quad \text{(for all } i = 1..k\text{)} \end{array}$$

3 Attribute Global Types

As we have shown in the previous section, the specification of ABP$_k$ conveniently exploits the constrained fork: by facing the problem with a "divide et impera" approach we were able to concentrate on the individual constraints that sending action types must respect, and compose them in an elegant and concise way.

Nevertheless, the specification of ABP$_k$ shown in Section 2 is not truly parametric in k, since we had to resort to the meta-notation with ellipses to represent sequences of arbitrary length k of types combined with the fork or sequence constructor, like in $msg_1{:}\ldots{:}msg_k{:}MM_k$. However ellipses do not work in practice, since for dynamic protocol verification the specification has to be executable, hence the framework as it is only allows specification of ABP_k for a fixed k, with the annoying shortcoming that different values for k require distinct specifications. To make an example, the constrained global type ABP_5 shown in Appendix B, although definitely more compact than the corresponding "plain" global type, shows redundancies and, more importantly, works only for the instantiation of ABP_k with $k = 5$. The three main reasons that hamper genericity are

1. the lack of a construct for iteration that, given a constrained global type τ and a construct $cn \in \{+, |, \cdot\}$, composes τ for a finite number of times n (given as a parameter), using cn;
2. the lack of parameters in both sending action types and in constrained global types;
3. the lack of conditions (for example, being in increasing order) on parameters.

Whereas the first problem can be solved by adding some syntactic sugar to the formalism, the second and third require a true extension of our constrained global types. In fact, the first problem can be tackled by adding to the language one "meta-construct" *fc* (for *finite composition*) that takes τ, the constructor *cn*, and the positive natural number n as inputs and generates the "normal" constrained global type (τ *cn* τ *cn* ... *cn* τ) (n times, hence $fc(\tau, cn, 1){=}\tau$). If we assume that a pre-processing is performed to translate any occurrence of $fc(\tau, cn, n)$ into the corresponding constrained global type, we need no changes to the transition rules to cope with it.

The last two problems, instead, require an extension of the formalism with parameters and conditions over them. Due to the similarity between this extension and attribute grammars (Knuth 1990), we named these parameters "attributes", and the resulting extended sending action types and constrained global types, "attribute sending action types" and "attribute global types" respectively. Along the lines of attribute grammars, an attribute global type is a constrained global type whose sending action types may have attributes, and that has been extended to provide contextual information by means of further attributes and conditions. Each attribute of a sending action type α has a domain of possible values and it may be bound to values from its domain in two ways:

– it may get its value when an actual sending action a occurs and verifies $a \in \alpha$ (synthesized attribute);

– it may get its value from the attributes of the global type it is part of (propagated attribute).

In a similar way, each attribute of an attribute global type τ has a domain of possible values and it may be bound to values from its domain in two ways:

– it may get its value from the attributes of attribute global types and attribute sending action types appearing in τ (synthesized attribute);

– it may get its value from the attributes of the global type it is part of (propagated attribute).

In our abstract syntax, an attribute sending action type is represented by $\alpha(attr_\alpha)$, where $attr_\alpha$ denotes a possibly empty sequence of distinct attributes, whereas an attribute global type is represented by $\tau(attr_\tau)[c]$, where $attr_\tau$ denotes a possibly empty sequence of distinct attributes, and c denotes a condition that must hold on $attr_\tau$. If $c = true$ then we use the abbreviated syntax $\tau(attr_\tau)$. In the definition of an attribute global type $\tau(attr)[c] = \tau'$, the attributes *attr* bind all occurrences in τ, to support attribute propagation. The only transition rules explicitly involving sending action types, seq-prod and seq-cons, are modified in the following way:

$$\text{(attr-seq-prod)}\,\frac{}{0, \alpha(attr_\alpha)^n{:}\tau \overset{a}{\rightarrowtail} \tau, n}\; {\scriptstyle a\in\alpha(attr_\alpha)}$$

$$\text{(attr-seq-cons)}\,\frac{}{n, \alpha(attr_\alpha){:}\tau \overset{a}{\rightarrowtail} \tau, n-1}\; \begin{array}{l}{\scriptstyle n>0}\\ {\scriptstyle a\in\alpha(attr_\alpha)}\end{array}$$

Usually, attributes in $attr_\alpha$ are taken from arguments of the actual sending action a. For example, assuming that i is a logical variable, we might state that $(Alice, Bob, tell, m(i)) \in msg(i)$ (resp. $(Bob, Alice, tell, a(i)) \in ack(i)$), with $i : Nat \ \wedge\ i \in [1..5]$, to make the definition of *msg* (resp. *ack*) in ABP_5 parametric in the index of the message (resp. acknowledge).

The transition rules for attribute constrained global types are those for constrained global types plus the following one, added to deal with types extended with attributes:

$$\text{(attr)}\,\frac{\tau \overset{a}{\rightarrowtail} \tau'}{\tau(attr_\tau)[c] \overset{a}{\rightarrowtail} \tau'}\quad c(attr_\tau)\text{ holds}$$

When we use $fc(\tau, cn, n)$ to replicate a global type τ with attributes, fresh attributes are created for each replication (under the constraint that the same attribute occurring in τ will be replaced by the same fresh attribute), to avoid bindings across the n instances of τ.

By exploiting *fc* and attributes, we are now able to express the ABP in the following way, parametric in the number k of message kinds with their respective acknowledges:

$$\begin{array}{l}
ABP'(k) = fc(MA(\_), |, k) | MM'(k, 0, \_)\\
MA(i) = msg^1(i){:}ack^0(i){:}MA(i)\\
MM'(k, p, c)[mod(k, p, c)] = msg(c){:}MM'(k, c, \_)
\end{array}$$

The definition of attribute sending action types $msg(i)$ and $ack(i)$ is the one given before, hence each different message kind (and its corresponding acknowledge) is represented by a natural index. Attribute k in both ABP' and MM' holds the number of different message kinds, i in *MA* denotes a specific message index, whereas p and c in MM' represent the previous and current message index, respectively.

Unnamed attributes _ are used when no specific value has to be associated with an attribute.

Let us analyze $ABP'(k)$ in detail. $ABP'(k)$ consists of a fork involving k replications of $MA(i)$ expressing the constraint that, for each i, the n-th occurrence of $msg(i)$ must precede the n-th occurrence of $ack(i)$, which in turn must precede the $(n+1)$-th occurrence of $msg(i)$. In each replication, i is replaced with a fresh attribute. This allows us to unify i with possibly different values in each branch. The finite fork is further composed with $MM'(k, 0, \_)$, whereas the definition of MM' uses the constraint $[mod(k, p, c)]$, which is a shortcut for the boolean condition

$$\begin{array}{c}
k : Nat \ \wedge\ p : Nat \ \wedge\ c : Nat \ \wedge\ p \in [0..k] \ \wedge\ c \in [1..k] \ \wedge\\
(p < k \Rightarrow c = p + 1) \ \wedge\ (p = k \Rightarrow c = 1)
\end{array}$$

The type $MM'(k, p, c)[mod(k, p, c)]$ specifies that indexes should increase until they reach k, and then they should start again from 1. Attributes k and p are propagated, the value of k is set once and for all when the type is defined, whereas the initial value of p is set to 0 in the definition of $ABP'(k)$.

When a transition from $MM'(k, 0, \_)$ to a new attribute global type takes place, condition $[mod(k, 0, c)]$ is evaluated for the first time. It holds if the synthesized attribute c, representing the index of the sending action of type $msg(c)$ allowed to be exchanged at that point of the protocol, can be assigned the value $0+1$. This means that the very first message exchanged during the protocol, must have sending action type $msg(1)$. Since $msg(1)$ in MM' must be consumed by a corresponding producer $msg^1(i)$ in one of the $MA(i)$ branches, from now on the index associated with that branch will be 1. When condition $[mod(k, p, c)]$ will be evaluated for the second time, p will be already bound to 1 and $[mod(k, p, c)]$ will hold if c can be bound to 2, meaning that the second sending action of type msg must have index 2. This will cause another $MA(i)$ branch to be bound to index 2, and so on. In the meanwhile, the acknowledge for $msg(1)$ could or could not have been exchanged, depending on how sending actions are shuffled.

Other examples. The extension of constrained global types with attributes allows us to express protocols with constraints that are very common in a MAS setting, such as constraints on the content of exchanged messages and deadlines. Let us consider, for example, the FIPA Iterated Contract Net Protocol (FIPA 2002b), ICNP for short, represented in Figure 3.

In the ICNP, one agent (the Initiator) takes the role of manager and solicits proposals by issuing a call for proposals *cfp* speech act, which specifies the task. Agents (Participants) receiving the call for proposals are viewed as potential contractors and are able to generate n responses. Among these, k are proposals to perform the task, specified as *propose* speech acts, and $n-k$ are denials (*refuse* speech act).

The Initiator evaluates the received k proposals: it may decide to terminate the iteration, accept p proposals, and reject the others or it may continue the iteration process by issuing a revised cfp to l Participants (and rejecting the remaining $k-l$). The process terminates when the Initiator either refuses all proposals and no new cfp is issued, or it accepts some proposals, or when all Participants refuse to propose.

The implementation of this protocol using an attribute global type is very intuitive. Given *pmax* a constant representing the maximum number of participants, and assuming that each agent in the system is named $Agent(k)$, $k \in [\mathit{1..pmax}]$, $cfp(init, i, lst)$ attribute sending actions type is defined as

$$(Agent(init), Agent(i), cfp, task) \in cfp(init, i, lst) \text{ if } init, i \in [1..pmax], task{:}Task, i \notin lst$$

stating that a cfp is correct if it has been sent by one agent in the system playing the role of Initiator to a different one (that is, not in lst) playing the role of Participant, and the content is a correct specification of a task. The other attribute sending action types can be defined in a very similar way.

As in the protocol shown in Figure 3, we distinguish between the first *cfp* that the Initiator sends to all the Participants, and the following ones which belong to interaction cycles going on only with a subset of the Participants that proposed; in this case the sending action type only requires that *init* is the sender and i the receiver, and that the content is a correct specification of a task.

The *ICNP* protocol is defined recursively, but at each step the value of its last attribute *part*, initially set to the constant *pmax*, is decreased by one: when *pmax* instances of the protocol have been combined with the fork operator, then a λ protocol is combined as the last element of the fork. The resulting type is defined as:

$$\begin{aligned}
&\mathit{ICNP}(\mathit{init}, \mathit{lst}, \mathit{part})[\mathit{part} > 0] = \\
&(\mathit{cfp}^{0}(\mathit{init}, i, \mathit{lst}) : \mathit{CYC}) | \mathit{ICNP}(\mathit{init}, [i|\mathit{lst}], \mathit{part} - 1) \\
&\mathit{ICNP}(\mathit{init}, \mathit{lst}, 0) = \lambda \\
&\mathit{CYC} = (\mathit{refuse}^{0}(i, \mathit{init}) : \lambda + \mathit{propose}^{0}(i, \mathit{init}) : \\
&\qquad\qquad (\mathit{cfp}^{0}(\mathit{init}, i) : \mathit{CYC} + \mathit{reject\_proposal}^{0}(\mathit{init}, i) : \lambda + \\
&\qquad\qquad \mathit{accept\_proposal}^{0}(\mathit{init}, i) : (\mathit{inform}^{0}(i, \mathit{init}) : \lambda + \mathit{failure}^{0}(i, \mathit{init}) : \lambda)))
\end{aligned}$$

ICNP is parametric in the index of the agent playing the Initiator role (attribute *init* is propagated and must be set when the global type is defined), in the list of agents that must not receive the first cfp (set to the singleton list with just the Initiator), and in the number of participants that still need to receive a cfp.

If we want to take deadlines into account, we just need to add a new attribute to *cfp* which expresses the deadline as an absolute time, and to distinguish between *propose* and *refuse* that arrive on time, and those that arrive late should be ignored.

For example, $(\mathit{Agent}(i), \mathit{Agent}(\mathit{init}), \mathit{propose}, (\mathit{task}, t)) \in \mathit{on\_time\_propose}(i, \mathit{init}, \mathit{dl})$ if $\mathit{task}{:}Task, t <= \mathit{dl}$, where t is the time when the sending action takes place. In a similar way we can define the *late_propose*, *on_time_refuse* and *late_refuse* sending action types, obtaining the attribute global type

$$\begin{aligned}
&\mathit{ICNP}_{\mathit{dl}}(\mathit{init}, \mathit{lst}, \mathit{part})[\mathit{part} > 0] = \mathit{cfp}^{0}(\mathit{init}, i, \mathit{lst}, \mathit{dl}) : (\mathit{CYC} + \mathit{LATE}))\ldots \\
&\mathit{CYC} = ((\mathit{on\_time\_refuse}^{0}(i, \mathit{init}, \mathit{dl}) : \lambda) + (\mathit{on\_time\_propose}^{0}(i, \mathit{init}, \mathit{dl}) : \ldots \\
&\mathit{LATE} = ((\mathit{late\_refuse}^{0}(i, \mathit{init}, \mathit{dl}) : \lambda) + (\mathit{late\_propose}^{0}(i, \mathit{init}, \mathit{dl}) : \lambda))
\end{aligned}$$

As a final requirement, we might want that each *cfp* and *propose* in the same *cfp-propose* interaction cycle, satisfy some "convergence constraint" such as that *cfp*s decrease some given parameter, and *propose*s increase another one, and the propose parameter must remain lower than the cfp one. Subtype CYC should be substituted with

$$\begin{aligned}
&\mathit{CYC}'(\mathit{cp}, \mathit{cc}, \mathit{pp}, \mathit{pc})[\mathit{dec\_inc}(\mathit{cp}, \mathit{cc}, \mathit{pp}, \mathit{pc})] = (\mathit{refuse}^{0}(i, \mathit{init}) : \lambda + \\
&\quad \mathit{propose}^{0}(i, \mathit{init}, \mathit{pc}) : (\mathit{reject\_proposal}^{0}(\mathit{init}, i) : \lambda + \\
&\qquad\qquad \mathit{accept\_proposal}^{0}(\mathit{init}, i) : (\mathit{inform}^{0}(i, \mathit{init}) : \lambda + \mathit{failure}^{0}(i, \mathit{init}) : \lambda) + \\
&\qquad\qquad \mathit{cfp}^{0}(\mathit{init}, i, \mathit{cc}) : \mathit{CYC}'(\mathit{cc}, \_, \mathit{pc}, \_)[\mathit{dec\_inc}(\mathit{cc}, \_, \mathit{pc}, \_)])
\end{aligned}$$

where the third parameter *pc* in *propose* (resp. *cc* in *cfp*) is a synthesized attribute representing the cost of the current proposal (resp. cfp), *cp* (resp. *pp*) is the previous value of the cost (the initial values should be fixed when the protocol is defined) and $\mathit{dec\_inc}(\mathit{cp}, \mathit{cc}, \mathit{pp}, \mathit{pc})$ is a shortcut for $\mathit{cc} = \mathit{cp} - 1 \ \wedge \ \mathit{pc} = \mathit{pp} + 1 \ \wedge \ \mathit{cc} > \mathit{pc}$.

4 Runtime verification of protocols in logic-based MASs

As introduced in Section 1, the motivation of our work is to provide a support for testing the conformance of a logic-based MAS to a given protocol represented as an attribute global type, overcoming some limitations in the expressive power and conciseness of other approaches.

To achieve our goal, one main requirement must be met by the MAS, namely, that there is a mechanism allowing messages directed from s to r, to be inspected and checked by a third agent m before they are actually sent. If no built-in support is given by the language/platform, an ad-hoc implementation of this requirement is always possible, and it should be kept less intrusive as possible. For the sake of presentation, in the remainder of this section we will assume that the agent language extends a basic Prolog core. Other logic-based languages supporting cyclic terms, term manipulation, meta-interpretation, and the ability to generate terms with fresh variables could take advantage of runtime verification of attribute global types as well.

A monitor agent m can be developed integrating the following pieces of code:

1. the Prolog representation of the attribute global type specifying the protocol to be tested, together with the clauses defining attribute sending action types;

2. the clauses defining the transition rules for attribute global types.

In the remainder of this section we will describe the two points above and we will discuss the experiments carried out with the Jason logic-based language.

Prolog representation of attribute global types. The representation of "plain" constrained global types in Prolog is straightforward and just translates in a machine readable format the syntax described in Section 2. In particular, λ is translated into `lambda`, $msg^n(attr)$, where $attr$ consists of $attr_1, \ldots, attr\_m$ is translated into `(msg(Attr1, ..., AttrM),N)`, $\cdot$ is translated into $*$. The other constructors have the same syntax, and we only need to take care of parentheses to avoid ambiguities. Constrained global types may be defined by unification equations and it is useful to name them by means of clauses whose head states the name of the protocol and whose body defines it. The condition $a \in \alpha$ is implemented by means of the `has_type/2` predicate, taking an actual message and a sending action type as arguments. To make an example, the clause

```
global_type(abp3, ABP3) :-
M1M2M3=lambda+(m1:(lambda+(m2:(lambda+(m3:(lambda+M1M2M3)))))),
M1A1=lambda+((m1,1):(lambda+((a1,0):(lambda+M1A1)))),
M2A2=lambda+((m2,1):(lambda+((a2,0):(lambda+M2A2)))),
M3A3=lambda+((m3,1):(lambda+((a3,0):(lambda+M3A3)))),
APB3=((M1M2M3|M1A1)|(M2A2|M3A3)).
```

along with facts like `has_type(msg(alice, bob, tell, m1),m1)`, defines the ABP$_3$.

Moving to the extensions described in Section 3, the *fc* constructor is defined as

```
fc(T, T1, C, N) :- N>1, copy_term(T1, Fresh), N1 is N-1, fc(T2, T1, C, N1),
T =.. [C, Fresh, T2].
fc(Fresh, T, _, 1) :- copy_term(T, Fresh).
```

and we assume that a call to `fc` is made to unify `T` with the corresponding composite type, before `T` is used in some unification equation. An attribute global type is represented by a fact

```
attrType(AttrTypeName, P(Attr), AttrGlobalType).
```

where `AttrTypeName` is a unique identifier of the attributed type, `P` is a metavariable standing for a predicate symbol that must hold on attributes `Attr` and `AttrGlobalType` is the definition of the type where attributes in `Attr` may appear.

By exploiting logical variables, the management of synthesized and propagated attributes comes for free in an easy and elegant way: those variables that are free when the type must be rewritten, are synthesized attributes, whereas those that are bound, are propagated ones.

Verification of protocol compliance. The runtime verification that a message a complies to the protocol, exploits the transition rules introduced in Section 2 and shown in Figure 1 to verify that given the current state $\tau 1$ of the protocol, and the actual a message, a new state $\tau 2$ exists such that $0, \tau_1 \overset{a}{\rightarrowtail} \tau_2, 0)$.

The transition rules for constrained global types have a very intuitive implementation into corresponding Prolog clauses defining a `next/5` predicate. The relationship between δ and `next` is $\delta(n, \tau 1, a, \tau 2, m) \iff$ `next(N, T1, A, T2, M)`. The implementation of `next/5` is given in Appendix A, together with predicate `empty/1` corresponding to ϵ.

The introduction of attributes to constrained global types just requires the addition of a clause for `next/5` implementing the rule

$$(\text{attr})\frac{\tau \overset{a}{\rightarrowtail} \tau'}{\tau(attr_\tau)[c] \overset{a}{\rightarrowtail} \tau'} \quad c(attr_\tau) \text{ holds}$$

The new clause is

```
/* attr */ next(M, attrType(Name, Constr, T1), A, T2, N) :-
   !, attrType(Name, Constr, T1), next(M, T1, A, T2, N), call(Constr).
```

While the Prolog code shown above can be reused in any logic-based agent language, maybe with minimal changes to comply with the language peculiarities, the way messages can be inspected heavily depends on the used agent framework and on the way the mechanism has been implemented. Depending on whether the compliance check succeeds or fails, different actions can be taken, which again depend on the adopted framework. The simplest one is that the monitor prevents the agent willing to send the non compliant message to actually send it, and that the protocol execution stops. More sophisticated prevention or recovery approaches can be implemented as well.

To show the feasibility of our approach, in the next paragraph we discuss our Jason implementation of the proposed monitoring mechanism.

Jason implementation of the runtime monitoring mechanism. Our Jason monitor keeps track of the runtime evolution of the protocol by saving its current state (which is an attribute global type), and checking that each message that a participant would like to send, is allowed by the current state. If so, the monitor allows the participant to send the message by explicitly sending an acknowledgment to it. On the other hand, if the monitor realizes that a violation would take place because of that message, no acknowledgment is sent to the participant and the monitor notifies the user that the implementation does not conform to the protocol. The monitor achieves its goal thanks to the Prolog code described in the previous subsections, with minor syntactic changes due to the lack of some built-in features, and to Jason plans for dealing with the various situations that may arise (protocol compliance, protocol violation, lack of exchanged messages for a given amount of time).

The Prolog code is shown in Appendix D, whereas the Jason plans are out of the scope of this paper and are omitted.

Currently, the Jason monitor is very strict: if a violation takes place, the entire protocol fails. In our recent research activity we performed a preliminary analysis of how it would be possible to design and implement systems able to prevent protocol violations, or to recover from them.

To show the potential of our approach, we have built a MAS with one agent playing the role of initiator in the ICNP, and a variable number of participants. The description of the ICNP protocol used by the monitor, shown in Appendix D, is the one discussed in Section 3. It integrates both deadlines and convergence constraints.

When running the MAS, we obtain console messages like those shown in Figures 4 and 5. For each message exchanged between two agents, the monitor prints out a messages like

```
[monitor]
Message msg(participant48,initiator,tell,m(propose(3,...,...),conv(initiator,0)))
leads from state fork(choice([choice([seq(sa(reject_proposal(initiator,participant50),0), ...
to state fork(choice([choice([seq(sa(reject_proposal(initiator,participant50),0),lambda),...
```

stating that the current message currently lead to a new state of the protocol,

```
[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
Message msg(participant11,initiator,tell,m(cfp(3,1365604497039,1365605493929),
conv(initiator,0))) received within protocol icnp
cannot be accepted in the current state fork(fork(choice([attrType(ontime,dec_inc
(participant11,initiator,27,_3824418,2,_3824419,1365605493929),_3824420),....
```

stating that the exchanged message does not comply with the protocol (in this case, a participant is issuing a *cfp* instead of any of the allowed messages),

```
[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
Message msg(participant1,initiator,tell,m(propose(3,1365604513461,1365605493929),
conv(initiator,0))) received but conversation conv(initiator,0) previously failed
```

stating that a message has been received, but the instance of the protocol (the conversation) it belonged to is no longer ongoing since it failed, and

```
[monitor]
*** WARNING ***
No progress for 4201 milliseconds
```

used for progress check.

The current implementation of the monitor is not optimized: the prototype is mainly meant at demonstrating the potential of our approach as well as its feasibility. Despite its simplicity, the monitoring of a Jason ICNP MAS consisting of more than 50 participants exchanging 22 messages in each conversation with the initiator required less than one minute on an Acer TravelMate 6293 with Intel Core 2 Duo P8400/2.26 GHz (Dual-Core) processor and 4 GB RAM. We also used SWI Prolog to generate simulated traces of protocol-compliant conversations in order to empirically validate the correctness of the formalization with respect to the actual protocol by manually inspecting a randomly selected subset of the traces. To give an idea of the feasibility and scalability of the generation process, generating the first 100 traces of the ABP_{20} protocol with 20 agents and traces long 100 required almost one hour and half, generating one single ICNP trace consisting of 1400 messages in a configuration with 700 different participants required a couple of minutes, and we were not able to generate any ICNP trace long 1000 with 300 participant as it took too long. Anyway, generating a trace of a given length is not the same problem as checking the protocol conformance on the fly. As SWI Prolog is very efficient, we are confident that a monitor agent implemented in SWI Prolog would be efficient and scalable. The code described in this paper can be downloaded here: `http://www.disi.unige.it/person/MascardiV/Software/globalTypes.html`.

5 Related and Future Work

Among the papers that inspired our work, the most crucial role was played by (Castagna et al. 2012). Whereas that paper focuses on "local" *session types*, which represent the *projections* of the global type on single entities, our work always takes a global perspective. Also, the interpretation of Castagna, Dezani-Ciancaglini, Padovani's global types is inductive (only interactions where the number of messages exchanged is finite can be modeled) whereas with attribute global types infinite interactions can be modeled as well. Constrained shuffle and attributes are not supported, and types cannot be recursive, hence the language is less expressive. Finally, we use global types for dynamic, rather than static, checking of multiparty interactions.

One of the first proposals born in the agent community for logic-based protocol representation and verification is (Endriss et al. 2003) whose authors identified different levels of conformance for a specific class of agents based on abductive logic programming and deterministic finite automata-based protocols. The two main differences between that seminal work and our proposal are the expressiveness of the language for representing protocols (attribute global types are more expressive than deterministic finite automata), and the type of verification carried out (a priori vs runtime). The paper (Giordano et al. 2007) describes a logical framework for specifying and verifying systems of communicating agents and interaction protocols based on Dynamic Linear Time Temporal Logic. The approach allows runtime verification of protocol compliance, but the whole trace of received messages needs to be kept in memory, while in our case no exchanged message needs to be saved. The paper (Torroni et al. 2009) describes how the SCIFF computational logic framework is used to provide the semantics of social integrity constraints. To model MAS interaction, expectation-based semantics specifies the links between the observed events and the expected ones. That approach shares with ours the possibility to perform runtime verification and to express attributes. However, it relies on the notion of expectations, that we do not formalize. None of the logic-based proposals above models protocol attributes in an explicit way. A work closer to ours is $\mathcal{RASA}$ (Miller and McBurney 2006), which combines constraints on attributes of the protocol and process algebra to model interaction protocols as first-class entities. The expressiveness of $\mathcal{RASA}$ is lower than that of our language as no fork operator is supported. The Lightweight Coordination Calculus LCC (Robertson 2004) is centered around constraints that must be respected to enforce social norms in a MAS. A LCC protocol specification consists of a set of Horn-like clauses whose head is the agent role and whose body is the definition of its behaviour when discharging that role. These clauses can be partitioned into subsets sharing the same head, representing the local point of view of that particular agent role. Although our protocols may be parametric into the agent's roles, we always describe the protocol from a global viewpoint.

The activities that we plan for our future work involve optimizing the verification mechanism, testing the expressiveness of attribute global types for formalizing some real protocol, and stress-testing the optimized monitor agent in order to perform a systematic empirical evaluation of the scalability of our approach. A formal analysis of the computational complexity of our approach will be made, to complement the empirical evaluation and finally assess the theoretic and practical usability of our language.

References

Ancona, D., Barbieri, M., and Mascardi, V. 2013a. Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In *SAC'13*. ACM, 1377–1379.

Ancona, D., Drossopoulou, S., and Mascardi, V. 2012. Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason. In *DALT X. Revised, Selected and Invited Papers*. LNAI, vol. 7784. Springer.

Ancona, D., Mascardi, V., and Barbieri, M. 2013b. Global types for dynamic checking of protocol conformance of multi-agent systems. Tech. rep., University of Genova, DIBRIS. Extended version of *D. Ancona, M. Barbieri, and V. Mascardi. Global Types for Dynamic Checking of Protocol Conformance of Multi-Agent Systems (Extended Abstract). In ICTCS 2012, pages 39-43, 2012.*

Bordini, R. H., Hübner, J. F., and Wooldridge, M. 2007. *Programming Multi-Agent Systems in AgentSpeak Using Jason*. John Wiley & Sons.

Carbone, M., Honda, K., and Yoshida, N. 2007. Structured communication-centred programming for web services. In *ESOP'07 (part of ETAPS 2007)*. LNCS, vol. 4421. Springer, 2–17.

Castagna, G., Dezani-Ciancaglini, M., and Padovani, L. 2012. On global types and multiparty session. *Logical Methods in Computer Science 8,* 1.

Endriss, U., Maudet, N., Sadri, F., and Toni, F. 2003. Logic-based agent communication protocols. In *ACL 2003*. LNCS, vol. 2922. Springer, 91–107.

FIPA. 2002a. FIPA ACL message structure specification. Approved for standard, Dec. 6^{th}, 2002.

FIPA. 2002b. FIPA iterated contract net interaction protocol specification. Standard Version 2002-12-06. Online at `http://www.fipa.org/specs/fipa00030/SC00030H.html`.

Giordano, L., Martelli, A., and Schwind, C. 2007. Specifying and verifying interaction protocols in a temporal action logic. *Journal of Applied Logic 5,* 2, 214 – 234.

Honda, K., Yoshida, N., and Carbone, M. 2008. Multiparty asynchronous session types. In *POPL 2008*. ACM, 273–284.

Huget, M.-P., Bauer, B., Odell, J., Levy, R., Turci, P., Cervenka, R., and Zhu, H. 2003. FIPA modeling: Interaction diagrams. Working Draft Version 2003-07-02. Online at `http://www.auml.org/auml/documents/ID-03-07-02.pdf`.

Jennings, N. R., Sycara, K. P., and Wooldridge, M. 1998. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems 1,* 1, 7–38.

Knuth, D. E. 1990. The genesis of Attribute Grammars. In *WAGA'90*. LNCS, vol. 461. Springer, 1–12.

Kowalski, R. A. and Sadri, F. 1999. From logic programming towards multi-agent systems. *Ann. Math. Artif. Intell. 25,* 3-4, 391–419.

Mayfield, J., Labrou, Y., and Finin, T. 1995. Evaluation of KQML as an agent communication language. In *ATAL'95*. Springer Verlag, 347–360.

Miller, T. and McBurney, P. 2006. Using constraints and process algebra for specification of first-class agent interaction protocols. In *ESAW'06*. LNCS, vol. 4457. Springer, 245–264.

Robertson, D. 2004. A lightweight coordination calculus for agent systems. In *DALT'04*. LNCS, vol. 3476. Springer, 183–197.

Torroni, P., Yolum, P., Singh, M. P., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., and Mello, P. 2009. Modelling interactions via commitments and expectations. In *Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models*. IGI Global.

Appendix A: transition rules for constrained global types and their Prolog implementation

The transition rules for constrained global types are shown in Figure 1, whereas the rules defining inclusion of λ are shown in Figure 2.

$$\text{(seq-prod)}\frac{}{0,\alpha^n{:}\tau \overset{a}{\rightarrowtail} \tau, n}\ {\scriptstyle a\in\alpha} \qquad \text{(seq-cons)}\frac{}{n,\alpha{:}\tau \overset{a}{\rightarrowtail} \tau, n-1}\ {\scriptstyle \begin{array}{l} n>0 \\ a\in\alpha \end{array}}$$

$$\text{(fork-both-l)}\frac{n_1,\tau_1 \overset{a}{\rightarrowtail} \tau_1', n_2 \quad n_2,\tau_2 \overset{a}{\rightarrowtail} \tau_2', n_3}{n_1,\tau_1|\tau_2 \overset{a}{\rightarrowtail} \tau_1'|\tau_2', n_3}\ {\scriptstyle n_2>0}$$

$$\text{(fork-both-r)}\frac{n_1,\tau_2 \overset{a}{\rightarrowtail} \tau_2', n_2 \quad n_2,\tau_1 \overset{a}{\rightarrowtail} \tau_1', n_3}{n_1,\tau_1|\tau_2 \overset{a}{\rightarrowtail} \tau_1'|\tau_2', n_3}\ {\scriptstyle n_2>0}$$

$$\text{(fork-l)}\frac{n_1,\tau_1 \overset{a}{\rightarrowtail} \tau_1', n_2}{n_1,\tau_1|\tau_2 \overset{a}{\rightarrowtail} \tau_1'|\tau_2, n_2} \qquad \text{(fork-r)}\frac{n_1,\tau_2 \overset{a}{\rightarrowtail} \tau_2', n_2}{n_1,\tau_1|\tau_2 \overset{a}{\rightarrowtail} \tau_1|\tau_2', n_2}$$

$$\text{(choice-l)}\frac{n_1,\tau_1 \overset{a}{\rightarrowtail} \tau_1', n_2}{n_1,\tau_1+\tau_2 \overset{a}{\rightarrowtail} \tau_1', n_2} \qquad \text{(choice-r)}\frac{n_1,\tau_2 \overset{a}{\rightarrowtail} \tau_2'}{n_1,\tau_1+\tau_2 \overset{a}{\rightarrowtail} \tau_2', n_2}$$

$$\text{(cat-l)}\frac{n_1,\tau_1 \overset{a}{\rightarrowtail} \tau_1', n_2}{n_1,\tau_1\cdot\tau_2 \overset{a}{\rightarrowtail} \tau_1'\cdot\tau_2, n_2} \qquad \text{(cat-r)}\frac{n_1,\tau_2 \overset{a}{\rightarrowtail} \tau_2', n_2}{n_1,\tau_1\cdot\tau_2 \overset{a}{\rightarrowtail} \tau_2', n_2}\ \epsilon(\tau_1)$$

Fig. 1. Rules defining δ

$$(\epsilon\text{-seq})\frac{}{\epsilon(\lambda)} \qquad (\epsilon\text{-choice-l})\frac{\epsilon(\tau_1)}{\epsilon(\tau_1+\tau_2)} \qquad (\epsilon\text{-choice-r})\frac{\epsilon(\tau_2)}{\epsilon(\tau_1+\tau_2)}$$

$$(\epsilon\text{-fork})\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1|\tau_2)} \qquad (\epsilon\text{-cat})\frac{\epsilon(\tau_1) \quad \epsilon(\tau_2)}{\epsilon(\tau_1\cdot\tau_2)}$$

Fig. 2. Rules defining inclusion of λ

Differently from other approaches (Castagna et al. 2012), global types are interpreted coinductively: for instance, the global type defined by $T = \alpha{:}T$ with $a \in \alpha$ denotes the set $\{a\,a\ldots\,a\,\ldots\}$ (that is, the singleton set containing the infinite sequence of sending action a), and not the empty set. Furthermore, whereas syntactically global types are regular trees, semantically their interpretation is not a regular language in general, for at least two reasons:

- the interpretation may contain strings of infinite length, as happens for the type $T = \alpha{:}T$;
- even when we remove the infinite strings from the interpretation of a type, what we get in general is not a regular language, because of recursion and the concatenation constructor.

Transition rules shown in Figure 1 have a one-to-one translation into clauses defining a `next/5` predicate, that implements the δ transition function.

```
/* seq-prod */   next(0,(AType,N):T,AMsg,T,N) :- has_type(AMsg, AType),!.
/* seq-cons */   next(N, AType:T1,AMsg,T1,M) :-
                 has_type(AMsg, AType), !, N > 0, M is N - 1.
/* fork-both-l */ next(N,T1|T2,A,T3|T4,P) :-
                  next(N,T1,A,T3,M), M > 0, next(M,T2,A,T4,P).
/* fork-both-r */ next(N,T1|T2,A,T4|T3,P) :-
                  !, next(N,T2,A,T3,M), M > 0, next(M,T1,A,T4,P).
/* fork-l */     next(N,T1|T2,A,T3|T2,M) :- next(N,T1,A,T3,M).
/* fork-r */     next(N,T1|T2,A,T1|T3,M) :- next(N,T2,A,T3,M).
/* choice-l */   next(N,T1+_,A,T2,M) :- next(N,T1,A,T2,M).
/* choice-r */   next(N,_+T1,A,T2,M) :- !, next(N,T1,A,T2,M).
/* cat-l */      next(N,T1*T2,A,T3*T2,M) :- next(N,T1,A,T3,M).
/* cat-r */      next(N,T1*T2,A,T3,M) :- !, empty(T1), next(N,T2,A,T3,M).
```

In a similar way, the five rules shown in 2 defining inclusion of λ can be defined by five clauses defining the `empty/1` predicate.

```
/* empty-seq */      empty(lambda) :- !.
/* empty-choice-l */ empty(T1+T2) :- empty(T1),!.
/* empty-choice-r */ empty(T1+T2) :- empty(T2).
/* empty-fork */     empty(T1|T2) :- !,empty(T1),empty(T2).
/* empty-cut */      empty(T1*T2) :- !,empty(T1),empty(T2).
```

Appendix B: representation of ABP$_5$ without attributes

The full specification of ABP$_5$ using constrained global types without attributes is the following:

$$\begin{array}{l}
ABP_5 = (MA_1|MA_2|MA_3|MA_4|MA_5)|MM_5\\
MA_1 = msg_1^1{:}ack_1^0{:}MA_1\\
MA_2 = msg_2^1{:}ack_2^0{:}MA_2\\
MA_3 = msg_3^1{:}ack_3^0{:}MA_3\\
MA_4 = msg_4^1{:}ack_4^0{:}MA_4\\
MA_5 = msg_5^1{:}ack_5^0{:}MA_5\\
MM_5 = msg_1{:}msg_2{:}msg_3{:}msg_4{:}msg_5{:}MM_5\\
(Alice, Bob, tell, m(1)) \in msg_1 \quad (Bob, Alice, tell, a(1)) \in ack_1\\
(Alice, Bob, tell, m(2)) \in msg_2 \quad (Bob, Alice, tell, a(2)) \in ack_2\\
(Alice, Bob, tell, m(3)) \in msg_3 \quad (Bob, Alice, tell, a(3)) \in ack_3\\
(Alice, Bob, tell, m(4)) \in msg_4 \quad (Bob, Alice, tell, a(4)) \in ack_4\\
(Alice, Bob, tell, m(5)) \in msg_5 \quad (Bob, Alice, tell, a(5)) \in ack_5
\end{array}$$

Appendix C: FIPA Iterated Contract Net Protocol

Figure 3 of this Appendix depicts the FIPA ICNP in FIPA-AUML (Huget et al. 2003).

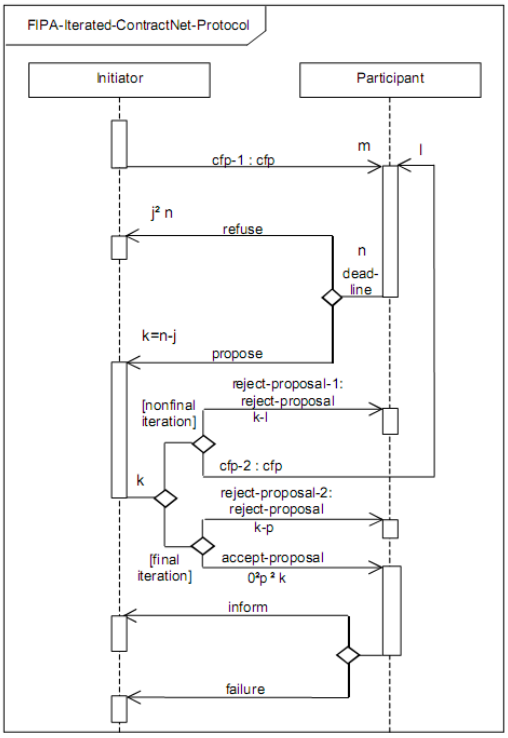


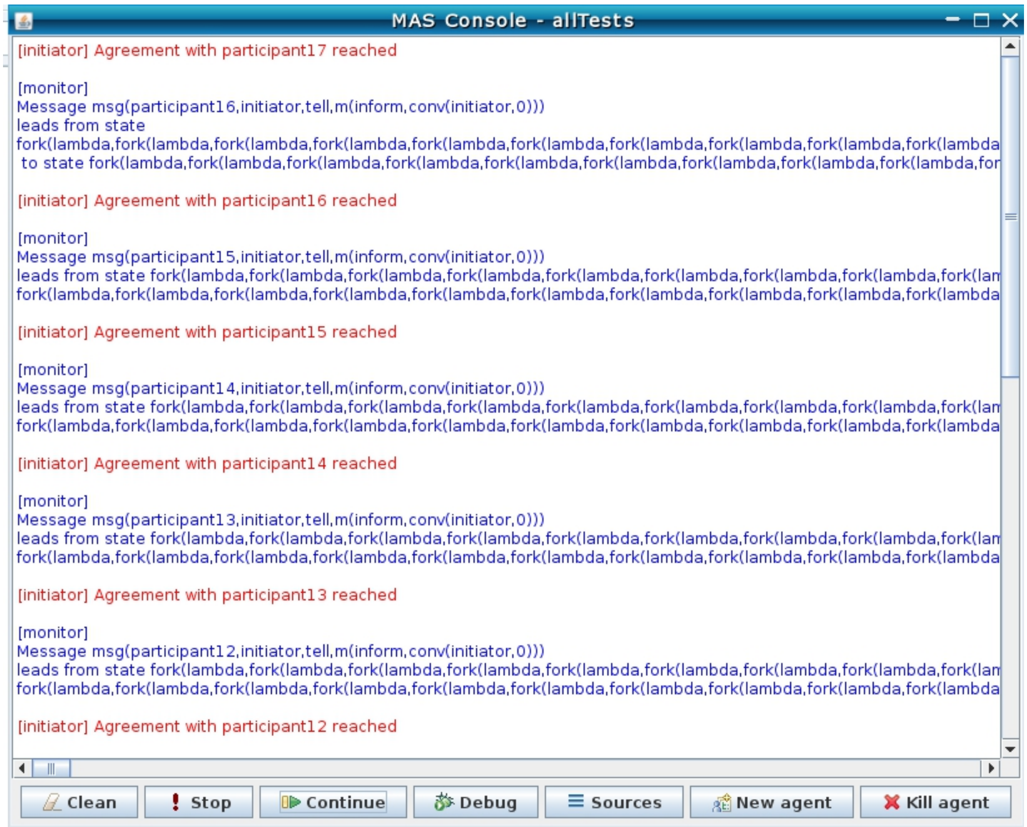
Fig. 3. FIPA Iterated Contract Net Protocol.

Appendix D: Jason implementation and experiments

The Prolog fragment of the Jason monitor's code is shown below.

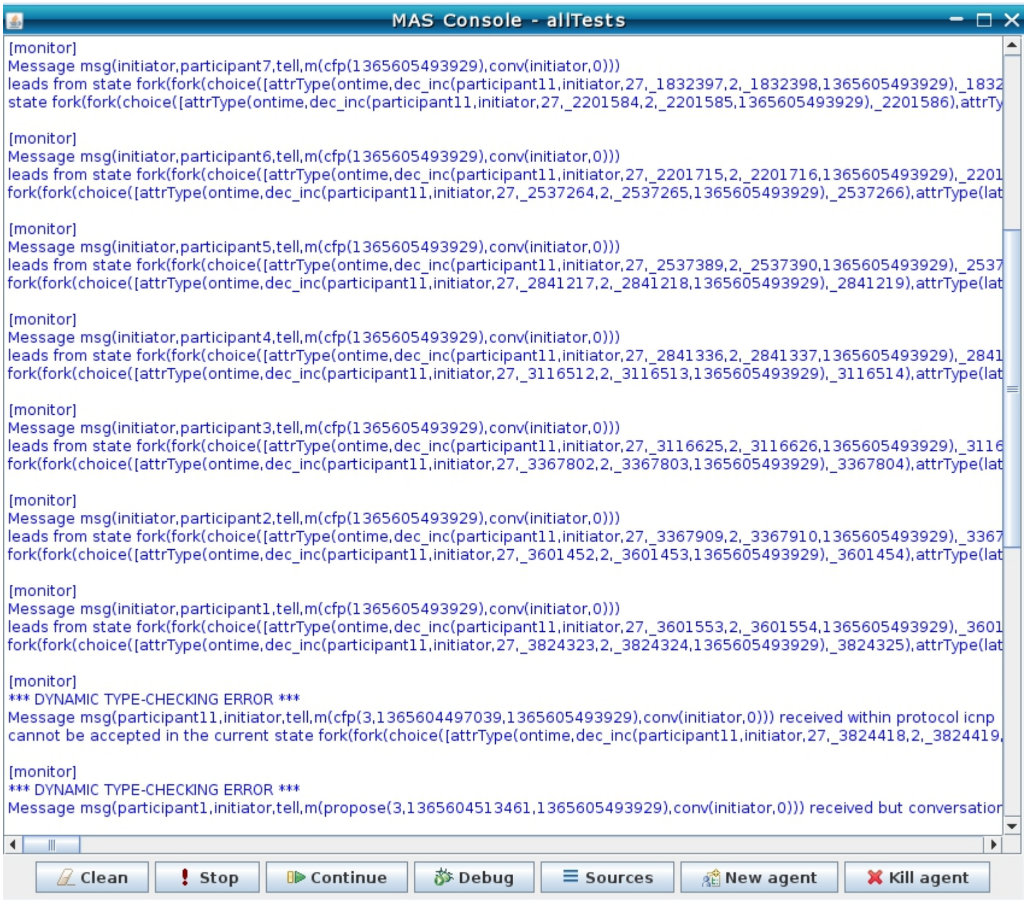
```
/*******************************************/
/*******************************************/
/* JASON MONITOR: PROTOCOL REPRESENTATION */
/*******************************************/
/*******************************************/

/*****************************************************************/
/**********                    ICNP                    *********/
/*****************************************************************/

/* Number of participants */
part_num(50).
/* Values in proposals and counter-proposals (for convergence checking) */
prev_cfp(27).
prev_propose(2).

/* ICNP protocol, involves initiator and participants */
global_type(icnp, T) :- // FIPA ICNP
   part_num(PartNum) &
   T = attrType(icnp, allDiff(_DL, _Init, PartNum, [], _), _).

attrType(icnp, allDiff(DL, Init, Current, List, X),
fork(
  seq(sa(cfp(Init, X, DL),0),
  choice([attrType(ontime, dec_inc(X, Init, PrevCFP, _, PrevPropose, _, DL), _),
          attrType(late, dl(DL), _)])),
attrType(icnp, allDiff(DL, Init, NewCurrent, [X|List], _), _))) :-
prev_cfp(PrevCFP) & prev_propose(PrevPropose) & Current > 0 & NewCurrent = Current-1.

attrType(icnp, allDiff(_DL, _Init, 0, _List, _X), lambda).
attrType(ontime, dec_inc(X, Init, _CFPprev, CFPcurr, _PROPprev, PROPcurr, DL),
choice([seq(sa(ontime_refuse(X, Init, DL),0),lambda),
 seq(sa(ontime_propose(X, Init, PROPcurr, DL),0),
   choice([
    choice([
     seq(sa(reject_proposal(Init, X),0),lambda),
     seq(sa(accept_proposal(Init, X),0),
        choice([
           seq(sa(inform(X, Init),0),lambda),
           seq(sa(failure(X, Init),0),lambda)
              ]))]),
    seq(sa(cfp(Init, X, CFPcurr),0),
       attrType(ontime, dec_inc(X, Init, CFPcurr, _, PROPcurr, _, noDl), _))
    ])
   )
  ])
 ).

attrType(late, dl(DL),
choice([seq(sa(late_refuse(DL),0),lambda),seq(sa(late_propose(DL),0),lambda)])).

/****************************************************************/
/**********             SENDING ACTION TYPES            *********/
/****************************************************************/

/* Sending action types for the ICNP protocol */
role(AgName, participant) :-
.term2string(AgName, NameStr) &
.substring("participant",NameStr, 0).

role(AgName, initiator) :-
.term2string(AgName, "initiator").

has_type(msg(Init, X, tell, m(cfp(DL), _Cid)), cfp(Init, X, DL)) :-
   role(Init, initiator) & role(X, participant).
has_type(msg(Init, X, tell, m(cfp(CFPcurr, DL), _Cid)), cfp(Init, X, CFPcurr)) :-
   role(Init, initiator) & role(X, participant).
has_type(msg(X, Init, tell,
   m(propose(PROPcurr, noDl), _Cid)), ontime_propose(X, Init, PROPcurr, noDl)) :-
   role(Init, initiator) & role(X, participant).
has_type(msg(X, Init, tell, m(propose(PROPcurr, CurrentTime, DL), _Cid)),
   ontime_propose(X, Init, PROPcurr, DL)):-
   role(Init, initiator) & role(X, participant) & CurrentTime <= DL.
has_type(msg(X, Init, tell, m(propose(PROPcurr, CurrentTime, DL), _Cid)),
   late_propose(DL)) :-
   role(Init, initiator) & role(X, participant) & CurrentTime > DL.
has_type(msg(X, Init, tell, m(refuse(CurrentTime, DL), _Cid)),
   ontime_refuse(X, Init, DL)) :-
   role(Init, initiator) & role(X, participant) & CurrentTime <= DL.
has_type(msg(X, Init, tell, m(refuse(CurrentTime, DL), _Cid)), late_refuse(DL)) :-
   role(Init, initiator) & role(X, participant) & CurrentTime > DL.
has_type(msg(Init, X, tell, m(reject_proposal, _Cid)), reject_proposal(Init, X)) :-
   role(Init, initiator) & role(X, participant).
has_type(msg(Init, X, tell, m(accept_proposal, _Cid)), accept_proposal(Init, X)) :-
   role(Init, initiator) & role(X, participant).
has_type(msg(X, Init, tell, m(inform, _Cid)), inform(X, Init)) :-
   role(Init, initiator) & role(X, participant).
has_type(msg(X, Init, tell, m(failure, _Cid)), failure(X, Init)) :-
   role(Init, initiator) & role(X, participant).

/****************************************************************/
/**********        CONSTRAINTS ON ATTRIBUTES            *********/
/****************************************************************/

dl(_).
dec_inc(_X, _Init, Cp, Cc, Pp, Pc, _) :- Cc = Cp-1 & Pc = Pp+1 & Cc > Pc.
allDiff(_, _, _, [], _).
allDiff(_, _, _, List, A) :- not(.member(A, List)).
```

```
/*********************************************/
/*********************************************/
/* JASON MONITOR: PROTOCOL-INDEPENDENT CODE  */
/*********************************************/
/*********************************************/

/****************************************************************/
/**********                 NEXT PREDICATE              *********/
/****************************************************************/

next(0,seq(sa(AType,N),T),AMessage,T,N) :- has_type(AMessage, AType).
next(M,seq(sa(AType),T1),AMessage,T1,N) :- has_type(AMessage, AType) & M > 0 & N = M - 1.
next(M,choice([T1,_]),A,T2,N) :- next(M,T1,A,T2,N).
next(M,choice([_,T1]),A,T2,N) :- next(M,T1,A,T2,N).
next(N,fork(T1,T2),A,fork(T3,T2),M) :- next(N,T1,A,T3,M).
next(N,fork(T1,T2),A,fork(T1,T3),M) :- next(N,T2,A,T3,M).
next(N,fork(T1,T2),A,fork(T3,T4),P) :- next(N,T1,A,T3,M) & M > 0 & next(M,T2,A,T4,P).
next(N,fork(T1,T2),A,fork(T4,T3),P) :- next(N,T2,A,T3,M) & M > 0 & next(M,T1,A,T4,P).
next(M,cat(T1,T2),A,cat(T3,T2),N) :- next(M,T1,A,T3,N).
next(M,cat(T1,T2),A,T3,N) :- empty(T1) & next(M,T2,A,T3,N).
next(M, attrType(Name, Constr, T1), A, T2, N) :-
attrType(Name, Constr, T1) &
next(M, T1, A, T2, N) &
Constr.

/****************************************************************/
/**********               EMPTY PREDICATE               *********/
/****************************************************************/

empty(lambda).
empty(choice([T1,_])) :- empty(T1).
empty(choice([_,T1])) :- empty(T1).
empty(fork(T1,T2)) :- empty(T1) & empty(T2).
empty(cat(T1,T2)) :- empty(T1) & empty(T2).


/****************************************************************/
/**********           FINITE CONSTRUCTORS               *********/
/****************************************************************/

finite_fork(T, T1, N) :-
   N>0 & gtu.copy_term(T1, Fresh) &
   N1 = N-1 & finite_fork(T2, T1, N1) &  T = fork(Fresh,T2).
finite_fork(Fresh, T1, 1) :- gtu.copy_term(T1, Fresh).

finite_choice(T, T1, N) :-
   N>0 & gtu.copy_term(T1, Fresh) &
   N1 = N-1 & finite_choice(T2, T1, N1) & T = choice([Fresh,T2]).
finite_choice(Fresh, T1, 1) :- gtu.copy_term(T1, Fresh).

finite_cat(T, T1, N) :-
   N>0 & gtu.copy_term(T1, Fresh) &
   N1 = N-1 & finite_cat(T2, T1, N1) &  T = cat(Fresh,T2).
finite_cat(Fresh, T1, 1) :- gtu.copy_term(T1, Fresh).

/****************************************************************/
/**********             TYPE_CHECK PREDICATE            *********/
/****************************************************************/

type_check(msg(S, R, P, C), NewState) :-
   current_state(LastState, CID) &
   next(0, LastState, msg(S, R, P, C), NewState, 0).
```



```
[initiator] Agreement with participant17 reached

[monitor]
Message msg(participant16,initiator,tell,m(inform,conv(initiator,0)))
leads from state
fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda
 to state fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,for

[initiator] Agreement with participant16 reached

[monitor]
Message msg(participant15,initiator,tell,m(inform,conv(initiator,0)))
leads from state fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lam
fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda

[initiator] Agreement with participant15 reached

[monitor]
Message msg(participant14,initiator,tell,m(inform,conv(initiator,0)))
leads from state fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lam
fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda

[initiator] Agreement with participant14 reached

[monitor]
Message msg(participant13,initiator,tell,m(inform,conv(initiator,0)))
leads from state fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lam
fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda

[initiator] Agreement with participant13 reached

[monitor]
Message msg(participant12,initiator,tell,m(inform,conv(initiator,0)))
leads from state fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lam
fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda,fork(lambda

[initiator] Agreement with participant12 reached
```

Fig. 4. Jason MAS: log of a compliant conversation.



```
[monitor]
Message msg(initiator,participant7,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_1832397,2,_1832398,1365605493929),_1832
state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_2201584,2,_2201585,1365605493929),_2201586),attrTy

[monitor]
Message msg(initiator,participant6,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_2201715,2,_2201716,1365605493929),_2201
fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_2537264,2,_2537265,1365605493929),_2537266),attrType(lat

[monitor]
Message msg(initiator,participant5,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_2537389,2,_2537390,1365605493929),_2537
fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_2841217,2,_2841218,1365605493929),_2841219),attrType(lat

[monitor]
Message msg(initiator,participant4,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_2841336,2,_2841337,1365605493929),_2841
fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3116512,2,_3116513,1365605493929),_3116514),attrType(lat

[monitor]
Message msg(initiator,participant3,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3116625,2,_3116626,1365605493929),_3116
fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3367802,2,_3367803,1365605493929),_3367804),attrType(lat

[monitor]
Message msg(initiator,participant2,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3367909,2,_3367910,1365605493929),_3367
fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3601452,2,_3601453,1365605493929),_3601454),attrType(lat

[monitor]
Message msg(initiator,participant1,tell,m(cfp(1365605493929),conv(initiator,0)))
leads from state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3601553,2,_3601554,1365605493929),_3601
fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3824323,2,_3824324,1365605493929),_3824325),attrType(lat

[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
Message msg(participant11,initiator,tell,m(cfp(3,1365604497039,1365605493929),conv(initiator,0))) received within protocol icnp
cannot be accepted in the current state fork(fork(choice([attrType(ontime,dec_inc(participant11,initiator,27,_3824418,2,_3824419,

[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
Message msg(participant1,initiator,tell,m(propose(3,1365604513461,1365605493929),conv(initiator,0))) received but conversatior
```

Fig. 5. Jason MAS: log of an non compliant conversation.