Global Types for Dynamic Checking of Protocol Conformance of Multi-Agent Systems
(Extended Abstract)

Davide Ancona, Matteo Barbieri, and Viviana Mascardi

DIBRIS, University of Genova, Italy
email: davide@disi.unige.it, matteo.barbieri@oniriclabs.com, mascardi@disi.unige.it

1 Introduction

Multi-agent systems (MASs) have been proved to be an industrial-strength technology for integrating and coordinating heterogeneous systems. However, due to their intrinsically distributed nature, testing MASs is a difficult task. In recent work [1] we have tackled the problem of run-time verification of the conformance of a MAS implementation to a specified protocol by exploiting global types on top of the Jason agent oriented programming language [2].

Global types [3,6,4] are a behavioral type and process algebra approach to the problem of specifying and verifying multiparty interactions between distributed components.

Our notion of global type closely resembles that of Castagna, Dezani, and Padovani, [4] except for two main differences: our types are interpreted coinductively, rather than inductively, hence they are possibly infinite sets of possibly infinite sequences of interactions between a fixed set of participants; in this way, protocols that must not terminate can be specified. Furthermore, we use global types for dynamic, rather than static, checking of multiparty interactions; errors can only be detected at run-time, but checking is simpler and more flexible, and no notion of projection and session type has to be introduced.

Global types can be naturally represented as cyclic Prolog terms (that is, regular terms), and their interpretation can be given by a transition function, that can be compactly defined by a Prolog predicate. With such a predicate, a Jason monitor agent can be automatically implemented to dynamically check that the message exchange between the agents of a system conforms to a specified protocol.

In this paper we continue our research in two directions: on the one hand, we investigate the theoretical foundations of our framework; on the other, we extend it by introducing a concatenation operator that allows a significant enhancement of the expressive power of our global types. As significant examples, we show how two non trivial protocols can be compactly represented in our framework: a ping-pong protocol, and an alternating bit protocol, in the version proposed by Deniélou and Yoshida [5]. Both protocols cannot be specified easily (if at all) by other global type frameworks, while in our approach they can be expressed by two deterministic types (in a sense made precise in the sequel) that can be effectively employed for dynamic checking of the conformance to the protocol.

2 Global type interpretation

A 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), representing the singleton set $\{\epsilon\}$ containing the empty sequence ϵ.
- $a{:}\tau$ (*seq*), representing the set of all sequences obtained by adding the sending action a at the beginning of any sequence in τ.
- $\tau_1 + \tau_2$ (*choice*), representing the union of the sequences of τ_1 and τ_2.
- $\tau_1|\tau_2$ (*fork*), representing the set obtained by shuffling the sequences in τ_1 with the sequences in τ_2 .
- $\tau_1 \cdot \tau_2$ (*concat*), representing the set of sequences obtained by concatenating any sequence of τ_1 with any sequence of τ_2.

As an example, $(((a_1{:}\lambda)|(a_2{:}\lambda)) + ((a_3{:}\lambda)|(a_4{:}\lambda))) \cdot ((a_5{:}a_6{:}\lambda)|(a_7{:}\lambda))$ denotes the set of message sequences

$$\left\{ \begin{array}{l} a_1a_2a_5a_6a_7, a_1a_2a_5a_7a_6, a_1a_2a_7a_5a_6, a_2a_1a_5a_6a_7, a_2a_1a_5a_7a_6, a_2a_1a_7a_5a_6, \\ a_3a_4a_5a_6a_7, a_3a_4a_5a_7a_6, a_3a_4a_7a_5a_6, a_4a_3a_5a_6a_7, a_4a_3a_5a_7a_6, a_4a_3a_7a_5a_6 \end{array} \right\}$$

Global types are regular terms, that is, can be cyclic: more abstractly, they are finitely branching trees (where nodes are type constructors) whose depth can be infinite, but that can only have a finite set of subtrees. A regular term can be represented by a finite set of syntactic equations, as happens, for instance, in Jason and in most modern Prolog implementations. For instance, the two equations $T_1 = (\lambda + (a_1{:}T_1)) \cdot T_2$, and $T_2 = (\lambda + (a_2{:}T_2))$ represent the following infinite, but regular, global types $(\lambda+(a_1{:}(\lambda+(a_1{:}\ldots)))) \cdot (\lambda+(a_2{:}(\lambda+(a_2{:}\ldots))))$ and $(\lambda + (a_2{:}(\lambda + (a_2{:}\ldots))))$, respectively.

To ensure termination of dynamic checking of protocol conformance, we only consider *contractive* (or *guarded*) types.

Definition 1. *A global type τ is* contractive *if it does not contain paths whose nodes can only be constructors in $\{+, |, \cdot\}$ (such paths are necessarily infinite).*

The type represented by the equation $T_1 = (\lambda+(a_2{:}T_1))$ is contractive: its infinite path contains infinite occurrences of $+$, but also of the $:$ constructor; conversely, the type represented by the equation $T_2 = (\lambda + ((T_2|T_2) + (T_2 \cdot T_2)))$ is not contractive. Trivially, every finite type (that is, non cyclic) is contractive.

The interpretation of a global type depends on the notion of transition, a total function $\delta{:}\mathcal{T} \times \mathcal{A} \rightarrow \mathcal{P}_{\mathit{fin}}(\mathcal{T})$, where $\mathcal{T}$ and $\mathcal{A}$ denote the set of contractive global types and of sending actions, respectively. As it is customary, we write $\tau_1 \xrightarrow{a} \tau_2$ to mean $\tau_2 \in \delta(\tau_1, a)$. Figure 1 (in the Appendix) defines the inductive rules for the transition function.

The auxiliary function ϵ, inductively defined in Figure 2 (in the Appendix), specifies the global types whose interpretation is equivalent to λ.

Proposition 1. *Let τ be a contractive type. Then $\tau \xrightarrow{a} \tau'$ for some a and τ' if and only if $\epsilon(\tau)$ does not hold.*

Note that the proposition above does not hold if we drop the hypothesis requiring τ to be contractive; for instance, if τ is defined by $T = T + T$, then neither $\epsilon(\tau)$ holds, nor there exist a, τ' s.t. $\tau \xrightarrow{a} \tau'$.

Proposition 2. *If τ is contractive and $\tau \xrightarrow{a} \tau'$ for some a, then τ' is contractive as well.*

The two propositions above ensures termination when the rules defined in Figures 1 and 2 are turned into an algorithm (implemented, for instance, in Prolog clauses, as done for Jason [1]).

Definition 2. *Let τ_0 be a contractive type. A* run *ρ for τ_0 is a sequence $\tau_0 \xrightarrow{a_0} \tau_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} \tau_n \xrightarrow{a_n} \tau_{n+1} \xrightarrow{a_{n+1}} \ldots$ such that*

- *either the sequence is infinite, or there exists k such that $\epsilon(\tau_k)$;*
- *for all τ_i, a_i, and τ_{i+1} in the sequence, $\tau_i \xrightarrow{a_i} \tau_{i+1}$ holds.*

We denote by $\alpha(\rho)$ the possibly 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 $\{\alpha(\rho) \mid \rho$ is a run for τ_0 $\}$ if τ_0 admits at least one run, $\{\epsilon\}$ otherwise.

Note that, differently from other approaches [4], global types are interpreted coinductively: for instance, the global type defined by $T = a{:}T$ denotes the set $\{a^\omega\}$ (that is, the singleton set containing the infinite sequence of sending action a), and not the empty set. Furthermore, whereas global types are regular trees, in general their interpretation is not a regular language, since it may contain strings of infinite length.

Finally, we introduce the notion of deterministic global type, which ensures that dynamic checking can be performed efficiently without backtracking.

Definition 3. *A contractive global type τ is deterministic if for any possible run ρ of τ and any possible τ' in ρ, if $\tau' \xrightarrow{a} \tau''$, $\tau' \xrightarrow{a'} \tau'''$, and $a = a'$, then $\tau'' = \tau'''$.*

3 Examples

In this section we provide two examples to show the expressive power of our formalism.

3.1 Ping-pong Protocol

This protocol requires that first Alice sends n (with $n \geq 1$, but also possibly infinite) consecutive ping messages to Bob, and then Bob replies with exactly

n pong messages. The conversation continues forever in this way, but at each iteration Alice is allowed to change the number of sent ping messages.

For simplicity we encode with *ping* and *pong* the only two possible sending actions; then, the protocol can be specified by the following contractive and deterministic global type (defined by the variable *Forever*):

$$\begin{aligned}&Forever = PingPong \cdot Forever\\&PingPong = ping{:}((pong{:}\lambda) + ((PingPong) \cdot (pong{:}\lambda)))\end{aligned}$$

3.2 Alternating Bit Protocol

We consider the Alternating Bit protocol, in the version defined by Deniélou and Yoshida [5]. Four different sending actions may occur: Alice sends msg1 to Bob (sending action msg_1), Alice sends msg2 to Bob (sending action msg_2), Bob sends ack1 to Alice (sending action ack_1), Bob sends ack2 to Alice (sending action ack_2). Also in this case the protocol is an infinite iteration, but 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.

We first show a non deterministic contractive type specifying such a protocol (defined by the variable $AltBit_1$).

$$\begin{aligned}&AltBit_1 = msg_1{:}M_2\\&AltBit_2 = msg_2{:}M_1\\&M_1 = (((msg_1{:}\lambda)|(ack_2{:}\lambda)) \cdot M_2) + (((msg_1{:}ack_1{:}\lambda)|(ack_2{:}\lambda)) \cdot AltBit_2)\\&M_2 = (((msg_2{:}\lambda)|(ack_1{:}\lambda)) \cdot M_1) + (((msg_2{:}ack_2{:}\lambda)|(ack_1{:}\lambda)) \cdot AltBit_1)\end{aligned}$$

Since the type is not deterministic, it would require backtracking to perform the dynamic checking of the protocol. The corresponding minimal deterministic type (defined by the variable $AltBit_1$) is the following:

$$\begin{aligned}&AltBit_1 = msg_1{:}M_2\\&AltBit_2 = msg_2 : M_1\\&M_1 = (msg_1 : A_2) + (ack_2 : AltBit_1)\\&A_1 = (ack_1 : M_1) + (ack_2 : ack_1 : AltBit_1)\\&M_2 = (msg_2 : A_1) + (ack_1 : AltBit_2)\\&A_2 = (ack_2 : M_2) + (ack_1 : ack_2 : AltBit_2)\end{aligned}$$

References

1. D. Ancona, S. Drossopoulou, and V. Mascardi. Automatic Generation of Self-Monitoring MASs from Multiparty Global Session Types in Jason. In *Declarative Agent Languages and Technologies (DALT 2012). Workshop Notes.*, pages 1–17, 2012.

2. R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons, 2007.
3. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *ESOP'07 (part of ETAPS 2007)*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
4. G. Castagna, M. Dezani-Ciancaglini, and L. Padovani. On global types and multiparty session. *Logical Methods in Computer Science*, 8(1), 2012.
5. P.-M. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP'12 (part of ETAPS 2012)*, LNCS. Springer, 2012.
6. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL 2008*, pages 273–284. ACM, 2008.

A Appendix

$$(\text{seq})\frac{}{a{:}\tau \xrightarrow{a} \tau} \qquad (\text{choice-l})\frac{\tau_1 \xrightarrow{a} \tau_1'}{\tau_1 + \tau_2 \xrightarrow{a} \tau_1'} \qquad (\text{choice-r})\frac{\tau_2 \xrightarrow{a} \tau_2'}{\tau_1 + \tau_2 \xrightarrow{a} \tau_2'}$$

$$(\text{fork-l})\frac{\tau_1 \xrightarrow{a} \tau_1'}{\tau_1|\tau_2 \xrightarrow{a} \tau_1'|\tau_2} \qquad (\text{fork-r})\frac{\tau_2 \xrightarrow{a} \tau_2'}{\tau_1|\tau_2 \xrightarrow{a} \tau_1|\tau_2'}$$

$$(\text{cat-l})\frac{\tau_1 \xrightarrow{a} \tau_1'}{\tau_1 \cdot \tau_2 \xrightarrow{a} \tau_1' \cdot \tau_2} \qquad (\text{cat-r})\frac{\tau_2 \xrightarrow{a} \tau_2'}{\tau_1 \cdot \tau_2 \xrightarrow{a} \tau_2'}\ \epsilon(\tau_1)$$

Fig. 1. Rules defining the transition function

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

Fig. 2. Rules defining global types equivalent to λ