Global Types for Agent Interaction Protocols (short paper)[*]

Federico Bergenti[1,*], Leonardo Galliera[2], Paola Giannini[2,*], Stefania Monica[3,*] and Riccardo Nazzari[2]

[1]*DISTI, Università di Parma, Italy*
[2]*DISIT & DISSTE, Università del Piemonte Orientale, Italy*
[3]*DISMI, Università di Modena e Reggio Emilia, Italy*

Abstract

We introduce an extension of global and local types tailored to the description of FIPA agent interaction protocols, formalize one of such protocols with these types. This paper is the first step of an ongoing project aimed at the definition of agent systems correctly implementing protocols by construction.

Keywords

Multiparty Sessions, Global Types, Agent Interaction Protocols, Agent Programming Languages

Introduction Dynamic systems are characterized by entities interacting in asynchronous ways. Software agents are software entities with the ability to [1]: react to environmental changes; take autonomous decisions and act accordingly to achieve their (explicit) goals; co-operate in loosely coupled Multi-Agent Systems (MASs). The interest in agent programming languages [2] dates back to the early proposals of agent technologies [3] and, since then, it has grown significantly. Agent programming languages represent an important research topic because they are recognized as important tools [4] to support *Agent-Oriented Software Engineering* (*AOSE*) [5] and applications [6]. Jadescript [7, 8] is an *Agent-Oriented Programming* (*AOP*) [3] language that was designed from participants of the current project [7] to support the construction of effective agents. The near-future development plans for Jadescript include a dedicated support for a collection of (agent) interaction protocols [9] standardized by the *Foundation for Intelligent Physical Agents* (*FIPA*, www.fipa.org), which is an IEEE Standards Committee established to promote interoperability among agents. FIPA specifies some general-purpose interaction protocols and FIPA-compliant agents are requested to support at least some of them. FIPA encourages designers and programmers to adopt these interaction protocols, which motivates the need for a dedicated support for them in Jadescript.

The seminal works on session types [10] and typestate have started a surge of research on behavioral type systems [11] for describing interaction protocols and enforcing various

ICTCS'24: Italian Conference on Theoretical Computer Science, September 11–13, 2024, Torino, Italy

[*]This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X) and has the financial support of the Università del Piemonte Orientale.

*Corresponding author.

federico.bergenti@unipr.it (F. Bergenti); paola.giannini@uniupo.it (P. Giannini); stefania.monica@unimore.it (S. Monica)

0000-0002-4756-4765 (F. Bergenti); 0000-0003-2239-9529 (P. Giannini); 0000-0001-6254-4765 (S. Monica)

properties including deadlock freedom [12]. A *multiparty session* (*MPS* for short) is an interaction among *participants/agents* communicating by exchanging messages [13, 14, 15]. The interaction is specified by a *global type* of the session. *Local* or *session* types may be retrieved as *projections* from the global type. Projectability, i.e., the existence of the projection on all participants, ensures that the protocol can be implemented. Session types give a a decoupled (i.e., distributed) view of a protocol from the perspective of each participant.

The global types of [15] have several limitations, that make them not suitable to specify the standard protocols of interaction between agents. In particular, a session involves a fixed set of participants, whose behaviour is individually specified when the session is first initiated: there is no notion of specifying the *behaviour for a class of participants* that share the same behaviour and no participant can *dynamically* (i.e., during an ongoing session) *leave* the interactions which are, however, basic requests for the FIPA protocols of interaction between agents.

A very expressive enhancement of global types was proposed in [16, 17]. Roles are defined as classes of local behaviours that an arbitrary number of participants can dynamically join and leave. This extension is very expressive. However, it is unrealistically implementable with the communication pattern of agent languages, since it requires some sort of centralized register handling the association between participants and roles. Other extensions, tailored to specific application domains, were proposed, most of them targeting a specific programming languages, [18, 19, 20]. Of the language independent ones we mention Pabble, [21, 22], in which multiple participants can be grouped in the same role and indexed and there is the possibility of changing participants in a role by *parameterisation*, and the one proposed in [23] to ensure good properties of the interactions in MPSs in spite of failures. It introduces the notions of *sub-sessions* and *role set* (similar to the roles of [16, 17]). Our proposal, inspired by the two mentioned extensions, is tailored to the goal of specifying FIPA protocols, in which often groups of agents are addressed by an agent mediating their interaction with the rest of the world. In the following, participant is a synonym for agent. We call these groups of participants *role sets* and their *coordinator* is the only participant interacting with them. The coordinator can broadcast a message to all the participants of the role set and there is a construct to execute a sub-protocol on all the participants in a role set. In addition to *projectability*, we give some *well-formedness restrictions* on the global types that are meant to enforce their realisability by Jadescript agents.

Global and Local Types with Role Sets In the following definitions we use the meta-variables: $\mathsf{p}, \mathsf{q}, \mathsf{r}$ for single *participants*; x, y, z for *participant variables* (in the scope of a `for`); p, q, r for either *participants or participant variables*; R for *role sets*; R for either single *participants or role sets*; Q for either *participants or participant variables or role sets*; ℓ for *labels (names) of messages* and S for *basic types* (`int`, `bool`, ...). A *global protocol* declaration, defined in Figure 1, specifies the *participants* and the *role sets* involved in the protocol and the associated *global type*. Each *role set* is coupled with a participant (from the declared ones) which is its *coordinator*. The body of the declaration is a *global type* G. First a choice of messages may be sent in addition to single participants also to all the participants in a role set. In this case we enforce the restriction that the sender be the coordinator of the role set. As usual we have a choice of different messages and after the communication the protocol continues as prescribed by the global type corresponding to the selected label. Recursion introduces a recursion variable

$$\texttt{global protocol}\ name(\overline{\mathsf{p}}; \overline{\langle \mathsf{R}, \mathsf{p} \rangle}) = \mathsf{G}$$

$$\begin{array}{rcl} \mathsf{G} & ::= & p \to Q\,\{\,\ell_i\langle S_i\rangle.\mathsf{G}_i\,\}_{i\in I} \\ & & \mid \mu X.\mathsf{G} \mid X \\ & & \mid \mathsf{End} \\ & & \mid\ \texttt{for}\ x : \langle \mathsf{R}, \mathsf{q}\rangle\ \mathsf{G}_1; \mathsf{G}_2 \\ & & \mid\ x \to \mathsf{q}\,\mathsf{Quit} \end{array}$$

$I \neq \emptyset$ and $\ell_h \neq \ell_k$ for $h \neq k$.

Figure 1: Global Protocols and Types

$$\texttt{local protocol}\ name\ \texttt{at}\ R(\overline{\mathsf{p}}; \overline{\langle \mathsf{R}, \mathsf{q} \rangle}) = \mathsf{T}$$

$$\begin{array}{rcl} \mathsf{T} & ::= & Q\,!\{\,\ell_i\langle S_i\rangle.\mathsf{T}_i\,\}_{i\in I} \\ & & \mid p?\{\,\ell_i\langle S_i\rangle.\mathsf{T}_i\,\}_{i\in I} \\ & & \mid \mu X.\mathsf{T} \mid X \\ & & \mid \mathsf{End} \\ & & \mid\ \texttt{for}\ x : \langle \mathsf{R}, \mathsf{q}\rangle\ \mathsf{T}_1; \mathsf{T}_2 \\ & & \mid\ x!\mathsf{Quit} \mid \mathsf{q}\,?\,\mathsf{Quit} \end{array}$$

Figure 2: Local Protocols and Types

X that can be used in its body to return to the beginning of G. As usual we assume recursion to be guarded. End stands for the end of the protocol, but also the end of a sub-protocol as we will see shortly. The `for` construct prescribes that the same protocol G_1 be executed by all the participants in the role set R. In G_1 the variable x denotes any participant in R. The semicolon preceding G_2 means that the coordinator q of R must have completed the protocol G_1 on all the participants in R before continuing as specified by G_2. So End occurring in G_1 does not mean the end of the whole interaction, but just of the sub-protocol G_1. In G_1 there cannot occur free recursion variables. We also impose the restriction that `for` cannot be nested. The FIPA protocols analysed, so far, can be formalized without nesting of `for`. However, eventually we would like to remove this restriction. Finally the last clause of the definition is used by a participant in a role set to exit from the protocol. This means that subsequent messages sent from the coordinator to the participants of its role set will not be sent to this participant. The highlighted constructs are the extension w.r.t. [15] .

The *local/session types* are the view of a protocol from the perspective of each participant. A *local protocol* declaration, , defined in Figure 2, specifies the *participants* and the *role sets* involved in the protocol from the point of view of a participant or role set and the associated *local/session.* The body of the declaration is a *local/session type* T.
For local types, we have the standard constructs: *choice of outputs* (sending a message) also called *internal choices*, *choice of inputs* (receiving a message) also called *external choices* and guarded recursion. Then we have `for` construct and the request and accept of the message for exiting from the interaction. The `for` construct can only occur in the local type of the coordinator of a role set and similar restrictions apply for the request and accept of the `Quit` message. We now show how a FIPA protocol is described with our types. The example illustrates also how the projection of the global type onto the participants and role sets is defined.

Global and Local Types for the Brokering Interaction Protocol In Figure 3 we formalize, using our global types, the FIPA Brokering Interaction and in Figures 4 and 5 we give the projections on its participants and role set. We use the Java-like syntax, coming from Scribble [24], which differs from our formal syntax mainly in the definition of the choice constructs (both for global and local types). The choice construct, e.g., line 4 of Figure 3 and lines 5 and 12 of Figure 4, specifies the *leader of the choice*, i.e., the sender of the communication. The branches in case of global types should start with a message from the leader to the same participant and in case of local types with the corresponding send or receive.

The participants of the protocol are the `initiator`, the `broker` and a number of `agent`s in

```
1  global protocol myBrokering(role initiator,role broker,roleset Subagents:broker){
2    forward(string) from initiator to broker.
3    choice at broker{
4      refuse() from broker to initiator.
5      stop() from broker to Subagents.End
6    } or {
7      agree() from broker to initiator.
8      findAgent(string) from broker to Subagents.
9      for agent:<Subagents,broker>{
10       choice at agent {
11         notPossible() from agent to broker.
12         QUIT() from agent to broker
13       } or {
14         canDo() from agent to broker.End
15       }
16     } ;
17     choice at broker {
18       failureNoMacth() from broker to initiator.
19       stop() from broker to Subagents. End
20     } or {
21       foundMatches() from broker to initiator.
22       inputData(string) from broker to Subagents.
23       for agent:<Subagents,broker>{
24           choice at agent {
25             result(string) from agent to broker.End
26           } or {
27             someError() from agent to broker.End
28           }
29         }
30       } ;
31       choice at broker {
32         replyFromSubagents(string) from broker to initiator.End
33       } or {
34         failureBrokering() from broker to initiator.End
35       }}}}
```

Figure 3: Global protocol for Brokering Interaction

the role set `Subagents` with the `broker` as their coordinator, line 1 of Figure 3. The interaction starts with the `initiator` asking, by sending a message `forward` to the `broker`, to `forward` its request to the subagents. We specified a simple `string` as the request, but more complex data structures maybe exchanged. After this there is a choice made by the `broker` that may decide to fulfil the request or to refuse it. So we have a choice, with leader the `broker` which branches, starting at lines 4 and 7, begin with a message sent from the `broker` to the `initiator`. In order to have the projection on the role set `Subagents`, whose participants will behave in a different manner in the two branches, the `broker` must send them a different message in two branches. Our projection use the *merge operator* of [25] to return a choice of inputs, that we can see in the local protocol of `Subagent` of Figure 5 starting at lines 5 and 7. Then the `broker` searches for the `agents` that may perform the request of the `initiator` using the subprotcol in the body of the `for`. For each agent the `broker` is waiting for a message from `agent` which may accept to perform the request, by sending `canDo()`, line 14, or refusing, by sending `notPossible()`, line 11, and leave the interaction with the `QUIT()` message, line 12. From now on the subsequent communications between the `broker` and the role set `Subagents` will not involve the `agents` that quit the protocol. We see the difference between `QUIT()` and `End`, lines 12 and 14 of Figures 3, when projecting on the role set `Subagents`. The communication at line 12 in the global protocol is projected to `QUIT()to broker`, line 10 of Figures 5, whereas `End` is replaced with the projection of the protocol following the `;`, lines 13-28 of Figures 5. The projection on the `initiator` of this subprotocol produces `End` since no communication involves the `initiator`, so for this participant the projection is just the projection of the rest of the protocol. The projection of a `for` produces a local `for` only for the `broker`, i.e., the

```
1  local protocol myBrokering at broker
2    (role initiator,roleset Subagents:broker)
3  {
4    forward(string) from initiator.
5    choice at broker{
6      refuse() to initiator.
7      stop() to Subagents.End
8    } or {
9      agree() to initiator.
10     findAgent(string) to Subagents.
11      for agent:<Subagents,broker>{
12       choice at agent {
13         notPossible() from agent.
14         QUIT() from agent
15       } or {
16         canDo() from agent.End
17       }
18     } ;
19     choice at broker {
20       failureNoMacth() to initiator.
21       stop() to SubAgent. End
22     } or {
23       foundMatches() to initiator.
24       inputData(string) to Subagents.
25       for agent:<Subagents,broker>{
26           choice at agent {
27             result(string) from agent.End
28           }  or {
29             someError() from agent.End
30           }
31         }
32       } ;
33       choice at broker {
34         replyFromSubagents(string) to
               initiator.End
35       } or {
36         failureBrokering() to initiator.End
37       }}}}
```

Figure 4: Projection on `broker`

```
local protocol myBrokering at initiator
  (role broker,roleset Subagents:broker)
{
  forward(string) to broker.
  choice at broker{
    refuse() from broker. End
  } or {
    agree() from broker.
    choice at broker {
      failureNoMacth() from broker. End
    } or {
      foundMatches() from broker.
      choice at broker {
        replyFromSubagents(string) from broker.End
      } or {
        failureBrokering() from broker.End
      }}}}
```

```
local protocol myBrokering Subagents:broker
  (role initiator,role broker)
{
  choice at broker{
    stop() from broker. End
  } or {
    findAgent(string) from broker.
    choice at agent {
      notPossible() to broker.
      QUIT() to broker
    } or {
      canDo() to broker.
      choice at broker {
        stop() from broker. End
      } or {
        inputData(string) from broker.
        choice at agent {
          result(string).End
         } or {
          someError() to broker.End
         }}}}}}
```

Figure 5: Projections on role set and coordinator

coordinator of the interaction, lines 11-18 of Figures 4. After the subprotocol the `broker`, if there are `agents` that responded positively to the request send a message `foundMatches()` to the `initiator`, sends the inputs of the request to the remaining `agents` and collects their responses via the subprotocol in the `for` the at lines 25-32. After that it sends the results, if there are any to the `initiator`. Again, in order to have the projection on the role set `Subagents`, whose participants will behave in a different manner in the two branches, the `broker` must a message to the `agents` also in the branch of failure to match, line 21.

Conclusion The work presented is a part of a larger research project, within the PRIN project "T-Ladies", [26], one of whose goals is to provide support for development/maintenance, automatic property verification/enforcement and bug detection of loosely connected, distributed, possibly heterogeneous interacting systems. Our aim is the "correct" implementation of interaction protocols between agents implemented in Jadescript, which is a language developed by members of the project. More specifically, we want to define Jadescript agents whose interaction behaviour follows, by construction, a given protocol. We use the MPS type methodology and define protocols with global types from which we derive by projection the local types of the agents. The implementation of an editor for the global and local types that *checks their projectability and well-formedness* can be found at [27]. We plan to translate these local types into Jadescript agents involved in the protocol and prove that resulting system have the properties of Session Fidelity and (possibly) Progress.

References

[1] S. Franklin, A. C. Graesser, Is it an agent, or just a program?: A taxonomy for autonomous agents, in: J. P. Müller, M. J. Wooldridge, N. R. Jennings (Eds.), Intelligent Agents III, Agent Theories, Architectures, and Languages, ECAI '96 Workshop (ATAL), Budapest, Hungary, August 12-13, 1996, Proceedings, volume 1193 of *Lecture Notes in Computer Science*, Springer, 1996, pp. 21–35. URL: https://doi.org/10.1007/BFb0013570. doi:10.1007/BFB0013570.

[2] C. Bădică, Z. Budimac, H.-D. Burkhard, M. Ivanovic, Software agents: Languages, tools, platforms, Computer Science and Information Systems 8 (2011) 255–298.

[3] Y. Shoham, Agent-oriented programming, Artificial Intelligence 60 (1993) 51–92.

[4] R. H. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, A. Ricci, A survey of programming languages and platforms for multi-agent systems, Informatica 30 (2006).

[5] F. Bergenti, M.-P. Gleizes, F. Zambonelli (Eds.), Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook, Springer, 2004.

[6] S. Distefano, G. Merlino, A. Puliafito, D. Cerotti, R. Dautov, Crowdsourcing and stigmergic approaches for (swarm) intelligent transportation systems, in: Q. Zu, B. Hu (Eds.), Human Centered Computing, Springer International Publishing, Cham, 2018, pp. 616–626.

[7] F. Bergenti, G. Caire, S. Monica, A. Poggi, The first twenty years of agent-based software development with JADE, Autonomous Agents and Multi-Agent Systems 34 (2020) 36:1–36:19.

[8] F. Bergenti, S. Monica, G. Petrosino, A scripting language for practical agent-oriented programming, in: Proceedings of the 8^{th} ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018) at ACM SIGPLAN Conference Systems, Programming, Languages and Applications: Software for Humanity (SPLASH 2018), ACM, 2018, pp. 62–71.

[9] S. Poslad, Specifying protocols for multi-agent system interaction, ACM Transactions on Autonomous and Adaptive Systems 2 (2007) 15:1–15:24.

[10] K. Honda, Types for dyadic interaction, in: E. Best (Ed.), CONCUR, volume 715 of *LNCS*, Springer, Heidelberg, 1993, pp. 509–523.

[11] D. Ancona, V. Bono, M. Bravetti, J. Campos, G. Castagna, P. Deniélou, S. J. Gay, N. Gesbert, E. Giachino, R. Hu, E. B. Johnsen, F. Martins, V. Mascardi, F. Montesi, R. Neykova, N. Ng, L. Padovani, V. T. Vasconcelos, N. Yoshida, Behavioral types in programming languages, Found. Trends Program. Lang. 3 (2016) 95–230. URL: https://doi.org/10.1561/2500000031. doi:10.1561/2500000031.

[12] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, L. Padovani, Global progress for dynamically interleaved multiparty sessions, Mathematical Structures in Computer Science 26 (2016) 238–302. URL: https://doi.org/10.1017/S0960129514000188. doi:10.1017/S0960129514000188.

[13] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: G. C. Necula, P. Wadler (Eds.), POPL, ACM Press, New York, 2008, pp. 273–284.

[14] L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, N. Yoshida, Global progress in dynamically interleaved multiparty sessions, in: F. van Breugel, M. Chechik (Eds.), CONCUR 2008, volume 5201 of *Lecture Notes in Computer Sci-*

ence, Springer, 2008, pp. 418–433. URL: https://doi.org/10.1007/978-3-540-85361-9_33. doi:10.1007/978-3-540-85361-9_33.

[15] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, Journal of ACM 63 (2016) 9:1–9:67.

[16] P. Deniélou, N. Yoshida, Dynamic multirole session types, in: T. Ball, M. Sagiv (Eds.), POPL, ACM, 2011, pp. 435–446. URL: https://doi.org/10.1145/1926385.1926435. doi:10.1145/1926385.1926435.

[17] P. Deniélou, N. Yoshida, A. Bejleri, R. Hu, Parameterised multiparty session types, Logical Methods in Computer Science 8 (2012). URL: https://doi.org/10.2168/LMCS-8(4:6)2012. doi:10.2168/LMCS-8(4:6)2012.

[18] D. Castro-Perez, R. Hu, S. Jongmans, N. Ng, N. Yoshida, Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures, Proc. ACM Program. Lang. 3 (2019) 29:1–29:30. URL: https://doi.org/10.1145/3290342. doi:10.1145/3290342.

[19] G. Cledou, L. Edixhoven, S. Jongmans, J. Proença, API generation for multiparty session types, revisited and revised using Scala 3, in: K. Ali, J. Vitek (Eds.), ECOOP 2022, volume 222 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 27:1–27:28. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.27. doi:10.4230/LIPICS.ECOOP.2022.27.

[20] N. Lagaillardie, R. Neykova, N. Yoshida, Stay safe under panic: Affine Rust programming with multiparty session types, in: K. Ali, J. Vitek (Eds.), ECOOP 2022, volume 222 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 4:1–4:29. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2022.4. doi:10.4230/LIPICS.ECOOP.2022.4.

[21] N. Ng, N. Yoshida, Pabble: Parameterised Scribble for parallel programming, in: PDP 2014, IEEE Computer Society, 2014, pp. 707–714. URL: https://doi.org/10.1109/PDP.2014.20. doi:10.1109/PDP.2014.20.

[22] N. Ng, N. Yoshida, Pabble: parameterised Scribble, Serv. Oriented Comput. Appl. 9 (2015) 269–284. URL: https://doi.org/10.1007/s11761-014-0172-8. doi:10.1007/S11761-014-0172-8.

[23] M. Viering, R. Hu, P. Eugster, L. Ziarek, A multiparty session typing discipline for fault-tolerant event-driven distributed programming, Proceedings of the ACM on Programming Languages 5 (2021) 1–30. URL: https://doi.org/10.1145/3485501. doi:10.1145/3485501.

[24] K. Honda, A. Mukhamedov, G. Brown, T. Chen, N. Yoshida, Scribbling interactions with a formal foundation, in: R. Natarajan, A. K. Ojo (Eds.), Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings, volume 6536 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 55–75. URL: https://doi.org/10.1007/978-3-642-19056-8_4. doi:10.1007/978-3-642-19056-8_4.

[25] N. Yoshida, L. Gheri, A very gentle introduction to multiparty session types, in: D. V. Hung, M. D'Souza (Eds.), Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings, volume 11969 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 73–93. URL: https://doi.org/10.1007/978-3-030-36987-3_5. doi:10.1007/978-3-030-36987-3_5.

[26] W. C. et alt., Typeful language adaptation for dynamic, interacting and evolving systems,

https://cazzola.di.unimi.it/t-ladies.html, 2024.
[27] L. Galliera, R. Nazzari, Jadescript, https://github.com/LMetal/Jadescript, 2024.