Multiparty-session-types Coordination for Core Erlang

Lavinia Egidi[1,*][a], Paola Giannini[2,†][b] and Lorenzo Ventura[1]
[1] *DiSIT, Università Piemonte Orientale, Alessandria, Italy*
[2] *DiSSTE, Università Piemonte Orientale, Vercelli, Italy*

Keywords: Multiparty Session Types, Delegation, Actor Model.

Abstract: In this paper, we present a formalization of multiparty-session-type coordination for a core subset of Erlang and provide a tool for checking the correctness of a system against the specification of its protocol. In Erlang *actors* are primitive entities, which communicate only through explicit *asynchronous message passing*. Our tool ensures that if an Erlang system is well typed, then it does not incur in deadlocks or have actors getting stuck waiting for messages that never arrive; moreover any message that is sent will eventually be read. The tool is based on *multiparty session types*, a formalism introduced to specify the structure of interactions and to ensure safety properties.

1 INTRODUCTION

Complex distributed systems are often described by processes interacting by exchanging messages according to predefined protocols. Such interactions are called *sessions*.

The methodology of programming with session types starts from a communication protocol, a *global type*, which specifies the overall behaviour of the system of interacting processes that we call *participants*. The local behaviour (a local type or *session type*) for each endpoint participant is then algorithmically obtained as the *projection* of the global type (Honda et al., 2008). Participants start a session by agreeing on a *private channel* and then interact *sending* and *receiving* messages on this channel.

From the beginning, session types were enhanced with the ability to *delegate* interactions from one participant in a session to another participant in a different session, see (Honda et al., 2008). Thereby, by using delegation, a participant involved in a session can at any point request that some participant in a different session conduct part of the interaction on its behalf. In (Honda et al., 2008; Coppo et al., 2015), delegation is realized by the *principal* sending its communication channel to the *delegate* which then uses it for its own communications with the other participants. At the level of global types this is described by sending a session type (a description of the interaction) from the principal to the delegate.

In this paper we describe participant behaviour using the *actor model*. Actors are entities with a unique identifier and a message queue called a mailbox and they react to incoming messages in various ways. We also use the new approach to describe sessions and delegation introduced in (Castellani et al., 2020), which extends the global types of (Castellani et al., 2019) where there are no channels representing sessions, and hence delegation cannot be explicitly modelled by passing channels over channels. Instead, delegation is modelled by enabling the *principal to temporarily lend its behaviour to the delegate*. This approach is appropriate for a setting in which participants are actors sending messages to other actors and reacting to messages present in their mailbox. This is the computational model of Erlang, (Erlang, 2022), where actors are primitive entities, implemented as lightweight processes which communicate only through explicit message passing. The lack of shared resources eliminates the risk of deadlocks. Our type checking proposal ensures moreover that
• actors will never remain in wait of unsent messages – in this case we call the system *lock-free*;
• all messages sent will be read – *no orphan messages*.
We demonstrate our proposal on a core version of Erlang that we call *Featherweight Erlang* (*FErlang* for

[a] https://orcid.org/0000-0002-9745-0942
[b] https://orcid.org/0000-0003-2239-9529
[*] This original research has the financial support of the Università del Piemonte Orientale.
[†] This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

Egidi, L., Giannini, P. and Ventura, L.
Multiparty-session-types Coordination for Core Erlang.
DOI: 10.5220/0011316600003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 532-541
ISBN: 978-989-758-588-3; ISSN: 2184-2833

short). In FErlang delegation is realized by the Erlang feature that an actor may register with the unique identifier of another and so may act in its behalf.

The main contributions of the paper are:
• we adapted the *theoretical framework* of (Castellani et al., 2020), in which process communication is synchronous and the operations of starting and ending delegation are atomic, to the asynchronous setting and the absence of atomicity for operations of starting and ending delegation;
• we implemented a tool that accepting an implementation of a multiparty session (written in a significant subset of the Erlang syntax) and a global type specifying its intended interaction protocol checks that the Erlang program executes the specified protocol.

The paper is organized as follows. In Section 2 we introduce a motivating example. In Section 3 we introduce the syntax of FErlang (Section 3.1), we present the session types and the type checking rules (Section 3.2), we define projections of global types on session types (Section 3.3) and propose a semantics, stating the properties of the typing system with respect to it (Section 3.4). A glimpse at the implementation is given in Section 4, related work described in Section 5. Finally in Section 6 we draw some conclusions and hint to future work [1].

2 CLIENT, SELLER AND BANK PROTOCOL

Here we revisit a typical example of delegation involving three participants: Client, Seller, and Bank, taken from (Castellani et al., 2020). In the protocol, the client and seller engage in a session in which they agree on the terms of a purchase. If the client decides to purchase, the seller delegates the processing of the client's credit card to the bank.

A global type for this protocol is in Figure 1. Global types are built from atomic *communications*, and *forward*, and *backward delegations*. A communication, $\mathsf{C} \to \mathsf{S}\colon \mathtt{title}\,\langle\mathtt{String}\rangle$, means that participant C sends the message `title` with payload of type `String` to participant S. Forward delegation, $\mathsf{S}\langle\!\langle\mathsf{B}$, indicates that participant S, the *principal*, delegates its communications to participant B, the *delegate*. Backward delegation, $\mathsf{B}\rangle\!\rangle\mathsf{S}$, says that participant B stops being a delegate and goes back to executing only its own communications.

[1]More examples, full definitions of typing rules and projection, and more details on the implementation can be found at http://people.unipmn.it/giannini/TesiVentura.pdf.

```
C → S: title ⟨String⟩.
S → C: price ⟨Int⟩.
  (  C → S: ok.
     S → B: +price ⟨Int⟩.
     S⟨⟨B
     S → C: pay.
     C → S: card ⟨String⟩.
     B⟩⟩S
       (  B → S: ok.S → C: date ⟨String⟩.End,
          B → S: ko.S → C: ko.End
       ),
     C → S: ko.End
  )
```

Figure 1: Global type for Client, Seller, and Bank protocol.

A global type is
• a choice of *branches*, enclosed in parentheses (_), starting with communications from the same sender and continuing with a global type or
• a forward/backward delegation followed by a global type or
• End indicating the end of the protocol and the value returned.
In our tool we also allow recursive protocols, omitted here for lack of space.

When the choice has a single branch we omit the parentheses.
Messages starting with a plus sign, such as `+price` are called *connecting* and are sent to participants who are optional in the protocol, i.e., may or may not participate.

The protocol is as follows.
- Firstly, Client sends a title to Seller, indicated by action $\mathsf{C} \to \mathsf{S}\colon \mathtt{title}\,\langle\mathtt{String}\rangle$;
- Seller responds by sending a price to Client, indicated by $\mathsf{S} \to \mathsf{C}\colon \mathtt{price}\,\langle\mathtt{Int}\rangle$;
- If the price is within Client's budget, then the following occurs:
  - Client sends a message ok to Seller;
  - Seller sends the price to B. The annotation + on the message `+price ⟨Int⟩` indicates that the receiver Bank connects to the session at that point.
  - Seller delegates its interaction to Bank, indicated by $\mathsf{S}\langle\!\langle\mathsf{B}$.
  - Bank impersonating Seller sends to Client a message instructing to proceed with the payment $\mathsf{S} \to \mathsf{C}\colon \mathtt{pay}$
  - Client sends its credit card number apparently to Seller but actually — thanks to delegation — to Bank;
  - Bank delegates back to Seller, indicated by $\mathsf{B}\rangle\!\rangle\mathsf{S}$.
  - If the payment went well, then (a) Bank sends a message ok to Seller and (b) Seller sends a `date` to Client (since the delegation has ended, this is a communication between the actual seller and the client) and the interaction ends;

- otherwise Seller sends message ko to Client, terminating the session.

• Otherwise, Client sends message ko to Seller, terminating the session.

Notice that, in the above protocol, the bank is never contacted in the case the buyer terminates the protocol by giving up the purchase. In other words, the bank is an optional participant. For this reason, it receives its first message via a connecting communication.

The local views of the participants in this protocol are described by the *session types* in Figure 2. The session type of each participant is obtained automatically by *projection* from the global type. The projection always succeeds under some mild constraints removing ambiguity and ensuring that delegations begin and end correctly.

Session type

$$\bigvee(\, \mathsf{p}_i!m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\,)^{i\in I} \tag{1}$$

specifies that a participant can send a message m_i to participant p_i. The message is either a simple message or a connecting message. The message is chosen out of a set of possible messages, and since the choice is made by the sender, it is called an *internal choice* and represented using the union symbol. The interaction continues following S_i if the message chosen was m_i to p_i. Alternatively, the participant can receive either a simple or a connecting message with

$$\bigwedge(\, \mathsf{p}?m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\,)^{i\in I} \tag{2}$$

If the participant receives message m_i from p, it will then continue the interaction following S_i. Which message is received is not a choice of the participant but it is dictated by the sender and it is therefore an *external choice*. We use the intersection symbol to emphasise that the process of the participant must have a branch for any message that can be sent to it. To avoid input races the messages must come from the same sender p. When I is a singleton (just one message can be sent/received to/from one participant) we drop the internal/external choice symbols and the parentheses and write simply $\mathsf{p}!m\langle\overline{\mathsf{T}}\rangle.\mathsf{S}$ and $\mathsf{p}?m\langle\overline{\mathsf{T}}\rangle.\mathsf{S}$. We call *input type* the first and *output type* the latter. We assume that choices are *non ambiguous*. That is, for internal choices for all $h,k \in I$ with $h \neq k$ either $\mathsf{p}_h \neq \mathsf{p}_k$ or $m_h \neq m_k$, for external choices for all $h,k \in I$ with $h \neq k$ $m_h \neq m_k$. We say that an external choice is *well formed* when $\{m_i\}^{i\in I}$ contains either all simple or all connecting messages.

In addition to exchanging messages, session types may specify *delegation actions*:

$$\mathsf{p}\langle\!\langle.\mathsf{S} \mid \rangle\!\rangle\mathsf{p}.\mathsf{S} \mid \langle\!\langle\mathsf{q}.\mathsf{S} \mid \mathsf{q}\rangle\!\rangle.\mathsf{S} \tag{3}$$

Delegation involves two participants: the *principal*, say p in what follows, and the *delegate*, say q. The delegate q acts on behalf of the principal p in the piece of code delimited by two matching delegation actions $\mathsf{p}\langle\!\langle$, dubbed *accept forward delegation* and $\rangle\!\rangle\mathsf{p}$, dubbed *request backward delegation*. Dually, the principal p suspends execution between two matching delegation actions $\langle\!\langle\mathsf{q}$, dubbed *request forward delegation*, and $\mathsf{q}\rangle\!\rangle$, dubbed *accept backward delegation*. We require that there be no action by the principal between $\langle\!\langle\mathsf{q}$ and $\mathsf{q}\rangle\!\rangle$. The previous property will be enforced by the fact that session types are projections of global types.

In Figure 2 we give the local types of Client, Seller and Bank. Note that, in the session type of Client, the client interacts always with the seller. Also observe that Seller does not perform any actions between delegating its behaviour to the bank and receiving it back. Since the client decides whether to purchase or not by sending the messages ok or ko respectively, the branches for the client are composed using the internal choice operator, indicating that at compile-time we do not know what decision the client will make. In contrast, we know how the seller will react in response to the decision the client makes — as it receives from the client either an ok or a ko message — and hence the branches of the seller are composed using external choice ($\bigwedge$). In the session type for the bank, the first action S?+price⟨Int⟩ indicates that the bank is ready to receive a message containing a price from the seller via a connecting communication. Then, on behalf of the seller, the bank sends a message instructing the client to pay and it receives a card number from the client, indicated by C?card⟨String⟩. Then, it returns control to the seller, as indicated by the construct $\rangle\!\rangle$S. Finally it sends to Seller a message saying whether the transaction was successful.

3 FERLANG AND ITS TYPE SYSTEM

3.1 FErlang Syntax

The syntax of FErlang is defined, in a continuation-passing style, in Figure 3. With X, a and pid we denote Erlang variables, atoms and process identifiers. With $\overline{X}$, $\overline{e}$ and $\overline{V}$ we mean sequences of X, e and V separated by comma, and with $\overline{F}$ a sequence of F. Production M describes a module declaration, where a after the keyword module indicates the module's name that is the actor's name. The module contains the definition of function start that takes no input and creates a process p using the built-in function spawn. The process p starts running the function specified by the second argument of spawn, in the module specified by the first argument, with $\overline{e}$ as arguments; p is registered with the actor's name. In the examples of

Client	Seller	Bank
S!title ⟨String⟩.	C?title ⟨String⟩.	S?+price ⟨Int⟩.
S?price ⟨Int⟩.	C!price ⟨Int⟩.	S⟪.
⋁(S!ok.	⋀(C?ok.	C!pay.
S?pay.	B!+price ⟨Int⟩.	C?card ⟨String⟩.
S!card ⟨String⟩.	⟪B.B⟫.	⟫S.
⋀(S?date ⟨String⟩.End,	⋀(B?ok.C!date ⟨String⟩.End,	⋁(S!ok.End,
S?ko.End	B?ko.C!ko.End	S!ko.End
),	),	)
S!ko.End	C?ko.End	
)	)	

Figure 2: Local types for Client (left), Seller (middle) and Bank (right).

M	$::=$	$\texttt{module}\ a\ \texttt{start}() \to$ $\texttt{register}(a, \texttt{spawn}(a, a, \overline{e})).\ \overline{F}$	module
F	$::=$	$\texttt{-spec}\ a(\overline{\mathsf{T}}) \to \mathsf{S}.\ a(\overline{X}) \to\ P$	fun def
P	$::=$	$a!m, P$	send
		$\mid \texttt{receive}\ \{a_i, a_i, \overline{X}_i\}\ \to\ P_i^{i\in I}$	receive
		$\mid \texttt{case}\ \{\overline{e}\}\ \texttt{of}\ \{\overline{V}_i\}\ \to\ P_i^{i\in I}$	case proc
		$\mid X = e, P$	let def
		$\mid e.$	expr
m	$::=$	$\{u, a, \overline{e}\}$	message
e	$::=$	$\ell \mid X \mid \texttt{self}() \mid e\, \texttt{op}\, e \mid a(\overline{e})$	
		$\mid \texttt{case}\ \{\overline{e}\}\ \texttt{of}\ \{\overline{V}_i\}\ \to\ e_i^{i\in I}$	case expr
ℓ	$::=$	$a \mid \texttt{pid} \mid n \mid s \mid \texttt{true} \mid \texttt{false}$	literals
u	$::=$	$a \mid X$	sender
V	$::=$	$\ell \mid X$	pattern

Figure 3: Syntax of FErlang.

Figures 4 and 5 the initial function for the processes is init().

Production F describes a function definition with name a, parameters $\overline{X}$ of type $\overline{\mathsf{T}}$, and body P of type S. FErlang functions have a single clause, i.e., take a list of variables $\overline{X}$, which is a pattern matching any input of the right length.

In Erlang, everything is an expression. In FErlang, to make the presentation clear, we distinguish between processes and expressions. Expressions denote literals, whereas processes denote sequences of interactions. When it comes to function calls, this distinction cannot be made in a context-free way. The body of a function is a process P describing the interactions of a participant with the others and at the end returning a value resulting from the evaluation of an expression (last production of P).

A process can consist in sending a message m to a (a *send* process) and then continuing with process P. A message m is a tuple composed as follows:
• a first element, u, that identifies the sender; it may be an atom (the name of the sender) or a variable meaning that a delegate of a principal sends the message,
• a second element, a, the label of the message, which identifies the kind of request,
• an optional payload, consisting of zero or more values of expressions $\overline{e}$.

A *receive process* is a sequence of clauses specifying a tuple m_i, with patterns, associated with a process P_i. The tuple starts with two atoms followed by zero or more distinct variables. If the tuple of the ith clause matches a message in the process queue, then the actor continues with process P_i.
For a *case process*, a sequence of expressions $\overline{e}$ are evaluated, the sequence of literals obtained, $\overline{\ell}$, are matched against the sequences of patterns $\overline{V}_i$ ($i \in I$), from the first to the last. If the literals $\overline{\ell}$ match one $\overline{V}_j$ the actor continues with process P_j. In FErlang, patterns can be only either a literal or a variable and in a sequence of patterns all the variables must be distinct. *Matching a pattern* literal ℓ against a literal (value) ℓ' succeeds only if $\ell = \ell'$, whereas a pattern variable X matches any literal (value) ℓ and binds X to ℓ. To avoid run-time errors, the matching should succeed for at least one sequence of patterns, that is the *patterns are exhaustive* for the sequence of expressions $\overline{e}$. This could be achieved by having a final branch with a sequence of distinct variables (a default choice).
The *let* construct binds the value of expression e to variable X and continues with P.
Finally, a process can be an expression, that could either denote the value returned or, in case of a function call, a process.

Expressions are defined by the production for e. We have literals (atoms, process identifiers, numbers, strings and booleans), variables, self(), which denotes the pid of the process in which it is executed, binary operations, function calls and case expressions. Case expressions differ from the case construct for processes in that the branches must be expressions. Note that function calls may appear both in a process (as a tail call) or in an expression. In the second case our type system will ensure that the body of the function has an expression type. The Erlang primitive functions register(a,e), unregister(a) and spawn$(a,a,\overline{e})$ can be used in FErlang in a restricted way: register(a,e) and unregister(a) are used in the delegate participant when implementing start and end of delegation, and spawn$(a,a,\overline{e})$ is used in the `start`() function of modules.

```
-module(seller).
start()->register(seller,
              spawn(seller,init,[])).
-spec init()->'#client?title<String>..'
init() ->
 receive
  {client,title,Title} ->
   Price = 50,
   client!{seller,price,Price},
    receive
     {client,ok}->
      bank!{seller,price,Price},
      bank!{seller,start_delegation,
          seller,self()},
      receive
       {bank,end_delegation} ->
        receive
         {bank, ok}->
          client!{seller,date,"
              22/10/2022"},'End';
         {bank, ko}->
          client!{seller,ko},'End'
        end
      end;
     {client,ko}->'End'
    end
 end.
```

Figure 4: Seller.

```
-module(bank).
start()->
  register(bank,spawn(bank,init,[])).
-spec init()->'#seller?price<Int>....'
init() ->
 receive
  {seller,price,Price}->
   receive
    {seller,start_delegation,Name,From}
        ->
     unregister(Name),unregister(bank),
     register(Name, self()),
     client!{Name, pay, Price},
     receive
      {client, card, CardNumber}->
       unregister(Name),
       register(Name,From),
       register(bank, self()),
       seller!{bank, end_delegation},
       case length(CardNumber)==16 of
        true->
         seller!{bank, ok},'End';
        false->
         seller!{bank, ko},'End'
        end
    end
   end
 end.
```

Figure 5: Bank.

Examples of FErlang code can be seen in Figures 4 and 5.

In the following, to make types and terms more readable, we use `p`, `q`, and `r` for participant names (which will also be module names), `m` for messages, and `f` for function names. All these are FErlang atoms.

3.2 Session Types and Typing Rules

Session Types. There are two kinds of types: *session types* S that specify the protocol of interaction of the participants, and *expression types* T that specify the kind of information exchanged between participants and returned at the end of a session. We call *process type* a session type which is not an expression type.

We already introduced the process type constructs and their meanings in Section 2, (1), (2) and (3). Here we give expression types T:

$$\begin{array}{rcl} \mathsf{T} & ::= & \mathtt{Int} \mid \mathtt{String} \mid \mathtt{Bool} \mid \mathtt{Atom} \mid \mathtt{Pid} \\ & & \mid \mathtt{Pid}\langle \mathtt{p} \rangle \mid \mathtt{AtS}\ \mathtt{End} \end{array}$$

For *expression types*, in addition to the standard Erlang types (first line) we have the types $\mathtt{Pid}\langle a \rangle$ and `AtS` used to type the code of a participant during delegation. In particular, a variable with type $\mathtt{Pid}\langle a \rangle$ is associated with the `pid` of the principal, and one with type `AtS` is associated with the principal's name. `End` is the type of any expression.

Type Checking Rules. In Figure 6 we show some selected representative typing rules of our type system.

We first summarize the notation we use. Let $\Gamma = \overline{X}{:}\overline{\mathsf{T}}$ be a mapping between variables and expression types and $\Delta = \overline{\mathtt{f}}{:}\overline{\mathsf{F}}$ where $\mathsf{F} = \overline{\mathsf{T}} \to \mathsf{S}$ is a mapping between function names (atoms) and their type. We call Γ the *variable environment*, or simply environment, and Δ the *function environment*. With $\Gamma(X) = \mathsf{T}$ we mean that $X{:}\mathsf{T} \in \Gamma$ and we say $\Gamma(X)$ undefined if $X{:}\mathsf{T} \notin \Gamma$ for all T. Similarly for Δ.

We define the following type checking judgments:
• $\Delta;\Gamma \vdash_{\mathtt{p}} P : \mathsf{S}$ saying that process P associated with participant $\mathtt{p}$ behaves as prescribed by session type S.
• $\Delta;\Gamma \vdash_{-} e : \mathsf{T}$ saying that expression e has type T.
• $\Gamma \vdash_{-} V_1 \dots V_n : \mathsf{T}_1 \dots \mathsf{T}_n \triangleright \Gamma'$ saying that patterns $V_1 \dots V_n$ *agree with types* $\mathsf{T}_1 \dots \mathsf{T}_n$. In case a V_i is a literal, then T_i must be its type, whereas a pattern variable matches any type. The environment Γ' is an extension of Γ obtained by adding associations between the V_i which are variables and the matched type T_i.
• $\mathsf{S} \leq \mathsf{S}'$ saying that S *is a subtype of* S' or S *is more specific than* S', meaning that a process (or expression) with type S can be used whenever one with type S' is required.

So finally, the rules from Figure 6, are:

- [Send]. Participant p has type $\mathsf{q}!m\langle\overline{\mathsf{T}}\rangle.\mathsf{S}$ if it sends a message to q and its continuation P has type S. The message sent must specify the sender, the message's label, and the payload. The sender can be either p itself or a variable that has type `AtS`, indicating that p has been delegated by the participant that will be bound to X to send the message to q (see Rule [Acpt-Fw-Del]). The message's label must be m, and the type of the expressions must be $\overline{\mathsf{T}}$.
- [Receive] Participant p has type $\bigwedge(\ \mathsf{q}?m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\ )^{i\in I}$ if receiving from q message m_i, $i\in I$, it branches to P_i that has type S_i. The patterns for the messages received must be tuples whose first two components are the sender's name q and a label m_i; the rest is a sequence of distinct variables $\overline{X}_i$ that will be bound to the values in the payload of the message. The pattern $\overline{X}_i$ matches any tuple of values of the right length. To type check the process P_i in the i-th branch, the association between the variables $\overline{X}_i$ and types $\overline{\mathsf{T}}_i$ are added to the environment Γ by the judgment $\Delta;\Gamma\vdash_{-}\overline{X}_i:\overline{\mathsf{T}}_i\triangleright\Gamma_i$ of Figure 6.
- [Case] Participant p has type $\bigvee(\ \mathsf{p}_i!m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\ )^{i\in I}$ if it is a `case` construct with $|I|$ branches and for all $i\in I$ the process P_i sends message m_i to p_i with payload of type $\overline{\mathsf{T}}_i$ and then process P_i has type S_i. Each branch is guarded by a pattern $\overline{V}_i$ whose type must be compatible with type $\overline{\mathsf{T}}$, the type of the expressions $\overline{e}$. To type check the process P_i, the association between the variables in pattern $\overline{V}_i$ and the corresponding types in $\overline{\mathsf{T}}$ are added to the environment. Even though FErlang syntax allows any process P_i in a branch, since the `case` construct must have an internal choice type, then each branch must have a send type. The type rule for the `case` construct for expressions, which is not shown, will require that each branch have the same expression type, as usual in functional languages.
- [Req-Fw-Del]: Participant p asks participant q to "take its role" in the protocol. To do so, p must send a `start_delegation` message to q. Then participant p must be such that the rest of its process P has type S.
- [Acpt-Fw-Del]: Participant q accepts to "take the role" of the principal p in the protocol. To do so, q must receive a `start_delegation` message from p and then:
  - disassociate both itself and the principal from their registered names, using the `unregister` function (this must be done before the following step because in Erlang the association between names and `Pid`s is a bijection);
  - register itself as having name p, using `register` and `self()` to get its own `Pid` and
  - the rest of its process Q must have type S.

  In the environment in which the process Q is type-checked, the variable X, which will be bound to the registered name of the principal, has type `AtS`, so that in Rule [Send] of Figure 6 we can check that the sender specified in a message sent by q, when q is acting as p's delegate, is indeed the principal p. Moreover, we associate Y to the type $\texttt{Pid}\langle\mathsf{p}\rangle$ so that in Rule [Req-Bw-Del] the registration can be correctly undone.
- [Req-Bw-Del]: Participant q "returns the control" of the protocol to p. To do so, q has to:
  - unregister the principal, denoted by X (which was associated with q's own `Pid`)
  - register itself with its original name q and the principal's `Pid` denoted by Y with its original name X (reverting to the state before the beginning of the delegation),
  - send an `end_delegation` message to the principal and
  - the rest of its process Q must have type S.

- [Acpt-Bw-Del] Upon receiving a request of ending delegation, the principal resumes its role.

Subtyping. For expression types we assume the subtyping: $\texttt{Pid}\langle a\rangle\leq\texttt{Pid}$ for all a and $\texttt{AtS}\leq\texttt{Atom}$. So we can use an expression of type $\texttt{Pid}\langle a\rangle$ whenever one of type `Pid` is required and similarly for `AtS` and `Atom`. Finally `End` is a supertype of all expression types (in Erlang this is denoted by `any()`). For process types we report the subtyping rule [Ext-$\leq$] for external choice, which is the most relevant:

$$\frac{\mathsf{S}_i\leq\mathsf{S}'_i\quad\forall i\in I}{\bigvee(\ \mathsf{p}?m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\ )^{i\in I\cup J}\leq\bigvee(\ \mathsf{p}?m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}'_i\ )^{i\in I}}$$

The more specific type may have more branches than the less specific and the common branches must start with sending the same message to the same participant with the same payload types. The type of the continuation of the corresponding branches must be in the same subtype relation. So the process having the more specific type receives all the messages as the less specific (could receive additional ones) and for those messages accepts more payload expressions. For internal choices the number of branches must be the same. We do not allow reducing the branches of internal choices, since this does not augment the set of typable sessions. Subtyping is used to match the inferred with the expected type of participants.

3.3 Projection

The main tool for enforcing the property that the FErlang multiparty session behave as expected by a protocol, and in particular be lock-free and have no orphan

$$\frac{\Delta;\Gamma \vdash_{\_} \overline{e} : \overline{\mathsf{T}} \quad \Gamma \vdash_{\mathtt{p}} P : \mathsf{S} \quad u = \begin{cases} X & \exists X\ \Gamma(X) = \mathtt{AtS} \\ \mathtt{p} & \text{otherwise} \end{cases}}{\Delta;\Gamma \vdash_{\mathtt{p}} \mathtt{q}!\{u, m, \overline{e}\}, P : \mathtt{q}!m\langle\overline{\mathsf{T}}\rangle.\mathsf{S}} \ [\textsc{Send}]$$

$$\frac{\Delta;\Gamma \vdash_{\_} \overline{X}_i : \overline{\mathsf{T}}_i \triangleright \Gamma_i \quad \Gamma_i \vdash_{\mathtt{p}} P_i : \mathsf{S}_i \quad \forall i \in I}{\Delta;\Gamma \vdash_{\mathtt{p}} \mathtt{receive}\ \{\mathtt{q}, m_i, \overline{X}_i\}\ \rightarrow\ P_i^{i\in I} : \bigwedge(\ \mathtt{q}?m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\ )^{i\in I}} \ [\textsc{Receive}]$$

$$\frac{\Delta;\Gamma \vdash_{\_} \overline{e} : \overline{\mathsf{T}} \quad \Gamma \vdash_{\_} \overline{V}_i : \overline{\mathsf{T}} \triangleright \Gamma_i \quad \Delta;\Gamma_i \vdash_{\mathtt{p}} P_i : \mathtt{p}_i!m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i}{\Delta;\Gamma \vdash_{\mathtt{p}} \mathtt{case}\ \overline{e}\ \mathtt{of}\ \{\overline{V}_i\}\ \rightarrow\ P_i^{i\in i} : \bigvee(\ \mathtt{p}_i!m_i\langle\overline{\mathsf{T}}_i\rangle.\mathsf{S}_i\ )^{i\in I}} \ [\textsc{Case}]$$

$$\frac{\Delta;\Gamma \vdash_{\mathtt{p}} P : \mathsf{S}}{\Delta;\Gamma \vdash_{\mathtt{p}} \mathtt{q}!\{\mathtt{p}, \mathtt{start\_delegation}, \mathtt{p}, \mathtt{self}()\}, P : \langle\!\langle \mathtt{q}.\mathsf{S}} \ [\textsc{Req-Fw-Del}]$$

$$\frac{\Delta;\Gamma, X : \mathtt{AtS}, Y : \mathtt{Pid}\langle\mathtt{q}\rangle \vdash_{\mathtt{q}} Q : \mathsf{S}}{\begin{array}{l}\Delta;\Gamma \vdash_{\mathtt{q}} \mathtt{receive}\ \{\mathtt{p}, \mathtt{start\_delegation}, X, Y\}\ \rightarrow \\ \qquad \mathtt{unregister}(\mathtt{q}), \mathtt{unregister}(X), \mathtt{register}(X, \mathtt{self}()), Q : \mathtt{p}\langle\!\langle.\mathsf{S}\end{array}} \ [\textsc{Acpt-Fw-Del}]$$

$$\frac{\Delta;\Gamma \vdash_{\mathtt{q}} P : \mathsf{S}}{\begin{array}{l}\Delta;\Gamma, X{:}\mathtt{AtS}, Y{:}\mathtt{Pid}\langle\mathtt{p}\rangle \vdash_{\mathtt{q}} \mathtt{unregister}(X), \mathtt{register}(\mathtt{q}, \mathtt{self}()), \\ \qquad\qquad \mathtt{register}(X, Y), X!\{\mathtt{q}, \mathtt{end\_delegation}\}, Q : \rangle\!\rangle \mathtt{p}.\mathsf{S}\end{array}} \ [\textsc{Req-Bw-Del}]$$

$$\frac{\Delta;\Gamma \vdash_{\mathtt{p}} P : \mathsf{S}}{\Delta;\Gamma \vdash_{\mathtt{p}} \mathtt{receive}\ \{\mathtt{q}, \mathtt{end\_delegation}\}\ \rightarrow\ P : \mathtt{q}\rangle\!\rangle.\mathsf{S}} \ [\textsc{Acpt-Bw-Del}]$$

Figure 6: Selected typing rules.

messages, is the definition of projection of a global type on a participant.

We define the *projection of a global type* G *on a participant* p, denoted $\mathsf{G} \upharpoonright \mathtt{p}$. For lack of space we give an informal description of projection by showing how the session types of Figure 2 are derived from the global type G of Section 2.

For an atomic communication followed by a global type G the projection on the message sender produces an output type on the message receiver, whereas the projection on the receiver produces an input type on the receiver; in both cases, followed by the projection of G on the same participant. Projection on a participant different from the sender or the receiver produces the projection of G on the participant. We can see (in Figures 1 and 2) that

$$\mathtt{C} \to \mathtt{S}\colon \mathtt{title}\ \langle\mathtt{String}\rangle.\mathsf{G}$$

projected on participant C yields $\mathtt{S!title}\ \langle\mathtt{String}\rangle$ and on S produces $\mathtt{C?title}\ \langle\mathtt{String}\rangle$ both followed by the projection of G. However, when projecting on Bank no action is generated. The projection of a choice of communications on the sender, say p, generates an internal choice where each branch contains the result of the projection of the initial communication followed by the projection of the corresponding branch. So the first action in each branch of p is a send. For the other participants the projection starts with a receive, and the receives in all branches produce a well formed external choice, as defined in Section 2. Going back to our example we see that the choice

$$(\ \mathtt{C} \to \mathtt{S}\colon \mathtt{ok}\ldots.\ ,\ \mathtt{C} \to \mathtt{S}\colon \mathtt{ko}\,.\mathtt{End}\ )$$

produces an internal choice of the Client and an external choice on the Seller. On the Bank it produces the initial receive $\mathtt{S?+price}\ \langle\mathtt{Int}\rangle$ which is a connecting communication.

The projection of a start delegation on the principal produces a *request forward delegation* followed by *delegation projection on the principal*, while the projection on the delegate produces an *accept forward delegation* followed by *delegation projection on the delegate*. If the participant is neither the principal nor the delegate, the projection of G ignores the delegation and continues with the standard projection previously described. The delegation projection on the principal is only defined if the communications that follow do not involve the delegate and we find an end delegation matching the starting one, which produces an *accept backward delegation*. The delegation projection on the delegate projects the action of the principal (as if they were of the delegate). Also for the delegate the projection is not defined if there is no end delegation. The projection of the end delegation is a *request backward delegation*. Consider

$$\mathtt{S}\langle\!\langle \mathtt{B}\ \mathtt{S} \to \mathtt{C}\colon \mathtt{pay}\,.\mathtt{C} \to \mathtt{S}\colon \mathtt{card}\ \langle\mathtt{String}\rangle.\mathtt{B}\rangle\!\rangle\mathtt{S}\ldots$$

The projection on Seller is, as it would be for any principal, $\langle\!\langle\mathtt{B.B}\rangle\!\rangle\ldots$ For the delegate the projection is $\mathtt{S}\langle\!\langle.\mathtt{C!pay.C?card}\ \langle\mathtt{String}\rangle.\rangle\!\rangle\mathtt{S}\ldots$ Notice that the actions of the principal are carried out by the delegate.

A *multiparty FErlang session* $\mathcal{M}$ is defined by a set of modules, p1.erl, ..., pn.erl, describing the processes of its participants and a module `main.erl` starting all the processes by calling the functions `start()` of `p1.erl`, ..., `pn.erl`.

Given a multiparty FErlang session $\mathcal{M}$, the global type G *is a type for* $\mathcal{M}$ if for all i, $1 \leq i \leq n$, if the module `pi.erl` starts with $\texttt{module}\ \mathtt{p}_i\ \texttt{start}() \to \texttt{register}(\texttt{spawn}(\mathtt{p}_i, \mathtt{f}_i, \overline{e})\ldots$, all its functions are well typed and, if S is the type for $\mathtt{f}_i(\overline{e})$, then $\mathsf{S} \leq \mathsf{G}\upharpoonright \mathtt{p}_i$. That is if, for every participant, the application of the corresponding initial function to its arguments behaves as prescribed by the projection of the global type G on the participant.

3.4 Semantics and Properties

The semantics of a multiparty FErlang session is given by the *set of traces of the execution of the* `main.erl` *module* where we consider only send and receive operations. Traces are obtained by the Erlang function `seq_trace` specifying `send` and `receive` as the tokens to be traced. We consider a trace semantics since Erlang actors are sequential processes.
The *trace of an execution* of a multiparty session, *tr*, is a (possibly empty) sequence of *actions* β where

$$\beta ::= \mathtt{p?q}\{\mathtt{m}, \ell_1, ..., \ell_n\} \mid \mathtt{p!q}\{\mathtt{m}, \ell_1, ..., \ell_n\}$$

The *player of* β, $\mathsf{play}(\beta)$, is the participant doing the action, that is $\mathsf{play}(\mathtt{p?r}\{\mathtt{m}, \mathtt{v}_1, ..., \mathtt{v}_n\}) = \mathsf{play}(\mathtt{p!q}\{\mathtt{m}, \mathtt{v}_1, ..., \mathtt{v}_n\}) = \mathtt{p}$. Given $tr = \beta_1 \cdots \beta_n$
• $tr[i] = \beta_i$ for i such that $1 \leq i \leq n$
• $tr[i..j] = \beta_i \cdots \beta_j$ for i and j such that $1 \leq i, j \leq n$

Definition 3.1. Let *tr* be a trace of length n, $tr[i]$ *matches* $tr[j]$ if, for $1 \leq i < j \leq n$,
• $tr[i] = \mathtt{p!q}\{\mathtt{m}, \mathtt{v}_1, ..., \mathtt{v}_n\}$, $tr[j] = \mathtt{q?p}\{\mathtt{m}, \mathtt{v}_1, ..., \mathtt{v}_n\}$
• and the number of actions $\mathtt{p!q}\{\_\}$ in $tr[1..(i-1)]$ is equal to the number of actions $\mathtt{q?p}\{\_\}$ in $tr[1..(j-1)]$.

We define the soundness of a trace for a global type as the property of executing the communications prescribed by the global type and doing it in the prescribed order.

Definition 3.2. The *trace tr of length n is sound for* G if the following hold:
• if $\mathsf{G} = (\ \mathtt{p} \to \mathtt{q}_i : \mathtt{m}_i\langle \mathsf{T}_i \rangle . \mathsf{G}_i\ )^{i \in I}$, then for some $i \in I$
 - there are j, k, $1 \leq j, k \leq n$ such that $tr[j] = \mathtt{p!q}_i\{\mathtt{m}_i, \ell_1, ..., \ell_n\}$ and $tr[k]$ matches $tr[j]$ and no $\mathtt{p!q}_i\{\_\}$ is in $tr[1..(j-1)]$ and
 - trace $tr[1..(j-1)] \cdot tr[(j+1)..(k-1)] \cdot tr[(k+1)..n]$ is sound for G_i,
• if $\mathsf{G} = \mathtt{p}\langle\!\langle \mathtt{q}.\mathsf{G}'$, then
 - there are j, k, $1 \leq j, k \leq n$ such that $tr[j] = \mathtt{p!q}\{\mathtt{start\_delegation}, \mathtt{p}, \mathtt{pid}\}$ and $tr[k] = \mathtt{q?p}\{\mathtt{start\_delegation}\}$ and
 - the number of actions $\mathtt{p!q}\{\_\}$ in $tr[1..(j-1)]$ is equal to the number of actions $\mathtt{q?p}\{\_\}$ in $tr[1..(k-1)]$ and
 - trace $tr[1..(j-1)] \cdot tr[(j+1)..(k-1)] \cdot tr[(k+1)..n]$ is sound for G',
• if $\mathsf{G} = \mathtt{q}\rangle\!\rangle \mathtt{p}.\mathsf{G}'$, then
 - there are j, k, $1 \leq j, k \leq n$ such that $tr[j] = \mathtt{p!q}\{\mathtt{end\_delegation}, \mathtt{p}, \mathtt{pid}\}$ and $tr[k] = \mathtt{q?p}\{\mathtt{end\_delegation}\}$ and
 - the number of actions $\mathtt{p!q}\{\_\}$ in $tr[1..(j-1)]$ is equal to the number of actions $\mathtt{q?p}\{\_\}$ in $tr[1..(k-1)]$ and
 - trace $tr[1..(j-1)] \cdot tr[(j+1)..(k-1)] \cdot tr[(k+1)..n]$ is sound for G',
• if G is an expression type and *tr* is empty.

A multiparty FErlang session $\mathcal{M}$ executes the protocol whose type is G if all its traces are sound for G. We can decide whether $\mathcal{M}$ executes the protocol whose type is G by type checking $\mathcal{M}$.

Theorem 3.3. Let $\mathcal{M}$=`p1.erl`, ..., `pn.erl`,`main.erl` be a multiparty FErlang session and G be a type. If G is a type for $\mathcal{M}$, then all finite traces of $\mathcal{M}$ are sound for G.

4 IMPLEMENTATION

The software consists of a JastAdd (Hedin, 2011) application that produces a type checker implemented in Java. JastAdd is a generator of syntax directed translators based on attribute grammars. Figure 7 shows the workflow of the tool on the example of Section 2.

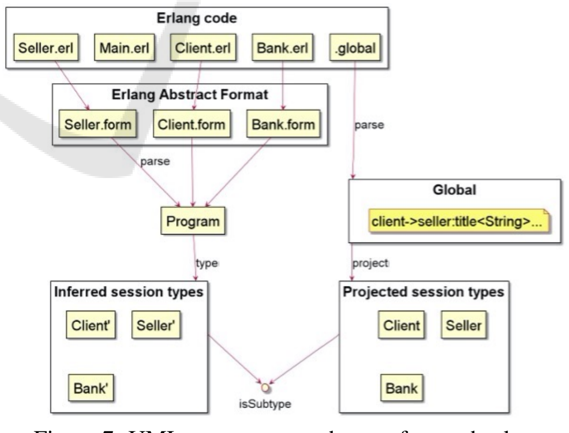

Figure 7: UML components schema of type checker.

Starting from the Erlang code, including a file named *.global* that contains the string representation of the global type, it checks that the global type is a type for the protocol.

The parse phase on the code generates the Abstract Syntax Trees (ASTs) from the `.form` files. JastAdd uses JFlex and Beaver as lexer and parser generators. The AST is specified, in a concise way in a `.ast` file, by giving the hierarchical relation between classes and their fields. From this specification JastAdd generates the corresponding Java classes. Finally the code for the computation of the attributes `type`, `project` and `isSubtype`, is specified in `.jrag` and `.jadd` files in a Java-like language (see Figure 8). From these specifications JastAdd generates methods, computing the values of the attributes, which are the core of the type checking, that are injected in the AST classes.

The most interesting attribute is the `type` attribute, associated with nodes that extend the `Process` class and return the inferred type for the Erlang expression. Its definition is not always straightforward since typing is not necessarily syntax driven; in particular, delegation makes the definition tricky. On the one hand, for a construct like `case` a union type is always inferred since the case is inherently an internal choice among its various branches.

On the other hand, a `receive` construct (for instance) must be dealt with looking inside the message that is received. The `type` attribute for the `Receive` behaves in a particular way when the message is a `start_delegation`. The fundamental reason is that a start delegation must be realized by a sequence of operations, disassociating first and rebinding later names and `Pids`. Therefore, before the `receive` is correctly typed, it must be checked that all operations are carried out and in the right sequence. This can be seen clearly in the snippet of the definition of the `type` attribute for the class `Receive` in Figure 8. We only report there part of the code for a `start_delegation`. After standard checks on the sender and on the syntax of the message (omitted in the figure), we check that the appropriate sequence of unregister and register operations follow the message receipt. In particular, notice that unregistrations must come before the registration of the delegate as the principal: lines 3 – 4 name the processes that follow the `receive`, and lines 9 – 10, together with lines 6 – 7, specify that the first two must be unregistrations. Notice that the order in which principal and delegate are unregistered is irrelevant (lines 9 – 10). After the unregistrations, the registration of the receiver of the message as principal is required (lines 8 and 11). Only after these checks (line 12) is the type `AcceptForwardDelegation` returned.

5 RELATED WORK

In (Mostrous and Vasconcelos, 2011), a session typing system for a featherweight Erlang is introduced, ensuring that the behavior of the process involved in a protocol is safe with respect to a protocol expressed by session types. The control that messages follow

```
1   if(getLabel().eq("start_delegation"
         )){
2    [...]
3     Process p = getActions(); Process
          p1 = getNextProcess(p);
4     Process p2 = getNextProcess(p1);
5     [...]
6     Unregister unreg1 = new
          Unregister(new List(new Atom(
          getModuleName())));
7     Unregister unreg2 = new
          Unregister( new List(d));
8     Register reg1 = new Register(new
          List(d,new Call(new Atom("
          self"),new List())));
9     boolean b1 = (p.eq(unreg1) || p.
          eq(unreg2));
10    boolean b2 = (p1.eq(unreg1) || p1
          .eq(unreg2));
11    boolean b3 = p2.eq(reg1);
12    if(b1 && b2 && b3 && !p.eq(p1)){
13        delegating().setName(d);
14        return new
              AcceptForwardDelegation(d
              ,getNextProcess(p2).type
              ());
15    }
16  }
```

Figure 8: Snippet of the definition of the `type()` attribute of `Receive`.

a prescribed pattern is realized by passing references (similar to channels) that uniquely identify sessions. Delegation is not allowed; in fact, authors of (Mostrous and Vasconcelos, 2011) also say that the very nature of Erlang makes delegation a delicate matter since communications are buffered, i.e., each process is co-located with its mailbox, and messages are addressed to process mailboxes. Moreover, no tool is provided for checking featherweight Erlang programs. We show that this problem can be overcome modelling delegation with unbinding of the principal's name and rebinding it to the delegate's, in a setting in which sessions are not represented by channels.

In (Neykova and Yoshida, 2017) the authors presented a framework for generating runtime monitors from a multiparty session type. Participants are implemented in Python based on a protocol for exchanging messages, Advanced Message Queuing Protocol (AMPQ), simulating an actor model. In (Fowler, 2016) an Erlang-based adaptation is proposed for the same runtime monitoring. Both works use Scribble to specify global types, a protocol based on the theory of multiparty session types, built as a Java-based toolchain, which incorporates tools for parsing, validating well-formedness, and local type projection. In these works there is no type checker which statically ensures that a specific protocol follows a global type. In (Harvey et al., 2021) and more in general in the Stardust project (Stardust, 2022), EnsembleS is introduced. This is an actor language leveraging multiparty session types to provide compile-time verification of safe dynamic runtime adaptation. The focus of their work is protocol adaptation in the face of actor failures.

We extended to FErlang the results of (Castellani et al., 2020), where the communication between processes is synchronous and the operation of starting and ending delegation is assumed to be atomic. The asynchronicity of FErlang, in contrast, allows more parallelism, but since requesting/accepting forward/backward delegation are non-atomic, periods exist in which the principal and/or the delegate are not registered, so messages sent to them could be lost. Our type checking system precisely ensures that this cannot happen if the actors follow the protocol prescribed by the session types obtained as projection from a global type: it ensures that the system is lock-free and has no orphan messages.

6 CONCLUSIONS

In this work, we present an Erlang implementation for multiparty protocols using a reduced, yet significant set of Erlang constructs (FErlang), including the communication primitives, send and receive, and constructs for delegation. For the description of delegation at the global type level we use the global types introduced in (Castellani et al., 2020) which include explicit primitives for starting and ending delegation. We formalized and implemented a projection from global types on session types that derives the expected behaviour of single participants from the global protocol. We defined a type system for FErlang and implemented a parser and type checker for it. The implementation was done using the meta compiler JastAdd, resulting in a system which can be easily extended to include new syntactic constructs (for FErlang, global and session types) and syntax directed translations.

For future work, in addition to using the tool on other protocols, we have several directions. On one side the current tool could be improved by tracking errors found during type checking; this could be done by defining in JastAdd further aspects to collect typing errors and showing them to the user. Moreover we could implement a GUI to help the user with the interaction. On the other hand, still leveraging on the JastAdd implementation we could also use the session type projection of a global type to generate a skeleton for the Erlang modules implementing the processes of the participants, as done in (Cutner and Yoshida, 2021) for Rust. The user could then customize that code relying on the fact that communications are correct.

ACKNOWLEDGEMENTS

We thank the anonymous referees for helpful comments.

REFERENCES

Castellani, I., Dezani-Ciancaglini, M., and Giannini, P. (2019). Reversible sessions with flexible choices. *Acta Informatica*, 56(7-8):553–583.

Castellani, I., Dezani-Ciancaglini, M., Giannini, P., and Horne, R. (2020). Global types with internal delegation. *Theoretical Computer Science*, 807:128–153.

Coppo, M., Dezani-Ciancaglini, M., Padovani, L., and Yoshida, N. (2015). A gentle introduction to multiparty asynchronous session types. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 146–178. Springer.

Cutner, Z. and Yoshida, N. (2021). Safe session-based asynchronous coordination in rust. In *COORDINATION*, volume 12717 of *LNCS*, pages 80–89. Springer.

Erlang (2022). Erlang doc. https://www.erlang.org. Accessed: 18-4-2022.

Fowler, S. (2016). An Erlang implementation of multiparty session actors. In *ICE*, volume 223 of *EPTCS*, pages 36–50.

Harvey, P., Fowler, S., Dardha, O., and Gay, S. J. (2021). Multiparty session types for safe runtime adaptation in an actor language. In *ECOOP*, volume 194 of *LIPIcs*, pages 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Hedin, G. (2011). *An Introductory Tutorial on JastAdd Attribute Grammars*, pages 166–200. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

Mostrous, D. and Vasconcelos, V. T. (2011). Session typing for a featherweight erlang. In *COORDINATION*, volume 6721 of *LNCS*, pages 95–109. Springer.

Neykova, R. and Yoshida, N. (2017). Multiparty session actors. *Logical Methods in Computer Science*, 13(1).

Stardust (2022). Stardust: Session Types for Reliable Distributed Systems. https://epsrc-stardust.github.io/. Accessed: 18-4-2022.