Lightening Global Types

Tzu-chun Chen
Dipartimento di Informatica, Università di Torino, Italy
chen@di.unito.it

Global session types prevent participants from waiting for never coming messages. Some interactions take place just for the purpose of informing receivers that some message will never arrive or the session is terminated. By decomposing a big global type into several light global types, one can avoid such kind of redundant interactions. Lightening global types gives us cleaner global types, which keep all necessary communications. This work proposes a framework which allows to easily decompose global types into light global types, preserving the interaction sequences of the original ones but for redundant interactions.

1 Introduction. Since cooperating tasks and sharing resources through communications under network infrastructures (e.g. clouds, large-scale distributed systems, etc.) has become the norm and the services for communications are growing with increasing users, it is a need to give programmers an easy and powerful programming language for developing applications of interactions. For this aim, Scribble [19], a communication-based programming language, is introduced building on the theory of global types [21,3]. A developer can use Scribble to code a global protocol, which stipulates any local endpoints (i.e. local applications) participating in it. The merits of coding global protocols, rather than just coding the local ones, are (1) giving all local participants a clear blue map of what events they are involved in and what purposes of those events and (2) making it easier and more efficient to exchange, share, and maintain communications plans (e.g. design of global protocols) across organisations. However, the tool itself cannot ensure an efficient communication programming. The scenario of a global communication can be very complicated so it becomes a burden for programmers to correctly code interactions which satisfy protocols (described by global types). At runtime, the cost for keeping all resources ready for a long communication and for maintaining the safety of the whole system can increase a lot.

For example, assume a gift requester needs a key (with her identity) to get a wanted gift. In order to get the key, she needs to get a guide, which is a map for finding the key. It is like searching for treasures step by step, where a player needs not be always online in one session for completing the whole procedure. Instead, the communication protocol can be viewed as separated but related sessions which are linked (we use calls to switch from a session to another). For example, let a session \( \mathcal{A} \) do the interactions for getting guide, and another session \( \mathcal{B} \) do the interactions for getting key with guide. Both guide and key are knowledge gained from these interactions. guide bridges \( \mathcal{A} \) and \( \mathcal{B} \) as it is gained in \( \mathcal{A} \) and used in \( \mathcal{B} \). The participant then uses key, if she successfully got it in \( \mathcal{B} \), to gain the wanted gift. Let session \( \mathcal{C} \) implements these final interactions.

As standard we use global types [21,3] to describe interaction protocols, adding a call command and type declarations for relating sessions. We call light global types the global types written in the extended syntax. Figure 1 simply represents the difference between viewing all interactions as one scenario and viewing interactions as three separated ones. As usual \( \text{store} \rightarrow \text{req} : \{ \text{yes3(str).end, no3().end} \} \) models a communication where role store sends to role req either the label yes3 and a value of type str or the label no3, and in both cases end finishes the interaction. The construct call \( \ell_b \) indicates that the interaction should continue by executing the session described by the light global type associated to the name \( \ell_b \).
The set of light global types associated to the names \( \ell_a, \ell_b \) and \( \ell_c \) describe the same protocol given by the global type \( G \). An advantage of lightening is to spare communications needed only to warn participants they will not receive further messages. We call such communications redundant ones. In the example we avoid the interactions \( \text{req} \rightarrow \text{store} : \text{no4}(\cdot) \) and \( \text{req} \rightarrow \text{issuer} : \text{no5}(\cdot) \text{req} \rightarrow \text{store} : \text{no6}(\cdot) \). Moreover, lightening prevents both local participants and global network from wasting resources, e.g. keeping online or waiting for step-by-step permissions.

Features of lightening become more clear by looking at the Scribble code implementing the global types of Figure 1, see Figure 2. The words in bold are keywords. In Scribble after the \textbf{protocol} keyword one writes the protocol’s name and declares the roles involved in the session, then one describes the interactions in the body. The left-hand side (LHS) of Figure 2 shows the Scribble code corresponding to the global type \( G \). Although the code is for a simple task, it is involved since there are three nested choices. On the contrary, the right-hand side (RHS) of Figure 2 clearly illustrates the steps for getting a gift in three small protocols (corresponding to the light global types associated to \( \ell_a, \ell_b, \ell_c \) respectively), which are separated but linked (by \textbf{run} and \textbf{at}) for preserving the causality.

The structure of the paper is the following. Section 2 introduces a framework of light global session types, which gives a syntax to easily compose a global type by light global types. Section 3 proposes a function for decomposing a general global type into light global types, e.g. it decomposes \( G \) into the types associated to \( \ell_a, \ell_b \) and \( \ell_c \). Section 4 proves the soundness of the function, and Section 5 discusses related and future works.

2 Syntax of (light) global types. The syntax for global types is standard:

\[
G ::= \ r_1 \rightarrow r_2 : \{l_j(S_j).G_j\}_{j \in J} \mid \mu t.G \mid t \ | \ \text{end} \quad \text{where} \quad S ::= \ \text{unit} \ | \ \text{nat} \ | \ \text{str} \ | \ \text{bool}
\]

The branching \( r_1 \rightarrow r_2 : \{l_j(S_j).G_j\}_{j \in J} \) says that role \( r_1 \) sends a label \( l_j \) and a message of type \( S_j \) to \( r_2 \) by selecting \( j \in J \) and then the interaction continues as described in \( G_j \). \( \mu t.G \) is a recursive type, where \( t \) is guarded in \( G \) in the standard way. end means termination of the protocol. We write \( l() \) as short for \( l(\text{unit}) \) and we omit brackets when there is only one branch.
Figure 2: Scribble for protocols getGift (LHS) and for getGuide, getKey, and getGift (RHS).
Light global types are global types extended with declarations and the construct call:

\[ D ::= \ell = L \quad L ::= \ r_1 \to r_2 : \{l_j(S_j).L_j\}_{j \in J} \ | \ \mu t.L \ | \ G \ | \ \text{call} \ \ell \]

Declarations associate names to light global types, for example in Figure 1 the name \( \ell_a \) is associated with the type

\[ \text{req} \to \text{map} : \text{req1} \ | \ \text{map} \to \text{req} : \{ \text{yes1} \ | \ \text{call} \ \ell_b, \ \text{no1} () . \text{end} \} \]

We denote by \( \emptyset \) the empty declaration.

The type call \( \ell \) prescribes that the interaction continues by opening a new session with light global type \( L \) such that \( \ell = L \) belongs to the set of current declarations. For example according to the declarations in Figure 1 call \( \ell_b \) asks to open a new session with type:

\[ \text{req} \to \text{issuer} : \text{req2} \ | \ \text{issuer} \to \text{req} : \{ \text{yes2} \ | \ \text{call} \ \ell_c, \ \text{no2} () . \text{end} \} \]

3 Lightening global types. This section describes a function for removing redundant interactions (Definition 1) from (possibly light) global types. It uses lightening, since it adds call constructors and declarations. In the next section we will show that the initial and final protocols describe the same non-redundant interactions.

We consider an interaction redundant when only one label can be sent and the message is not meaningful, i.e. it has type unit and it does not appear under a recursion. More precisely, by defining light global contexts (without recursion) as follows:

\[ C ::= [ \ | \ r_1 \to r_2 : \{l_j(S_j).G_j,l(S).C\}_{j \in J} \]

we get:

**Definition 1** (Redundant interaction). The interaction \( r_1 \to r_2 : l() \) is redundant in

\[ C[r_1 \to r_2 : l() . L] . \]

Interactions sending only one label and with messages of type unit are needed inside recursions when they terminate the cycle. For example the interaction \( r_2 \to r_3 : \text{stop}() \) cannot be erased in the type

\[ \mu t.r_1 \to r_2 : \{\text{goon} (\text{int}).r_2 \to r_3 : \text{goon} (\text{int}).t, \text{stop}().r_2 \to r_3 : \text{stop}().\text{end}\} \]

We define \( \mathbb{L}(L) \) as a function for removing redundant interactions in \( L \) by decomposing \( L \) into separated light global types. The basic idea is that, in order to erase redundant communications, the roles which are the receivers of these communications must get the communications belonging to other branches in separated types. Therefore the result of \( \mathbb{L}(L) \) is a new light global type and a set of declarations with fresh names.

The function \( \mathbb{L}(L) \) uses the auxiliary function \( L \downarrow x \) which searches inside the branches of \( L \) the first communications with receiver \( x \) and replaces these communications (and all the following types) by calls to newly created names, which are associated by declarations to the corresponding types. Therefore also the result of \( L \downarrow x \) is a new light global type and a set of declarations with fresh names.
Definition 2 (The function $\downarrow$). The function $L \downarrow r$ is defined by induction on $L$:

$$\downarrow \text{end} = (\text{end}, \emptyset)$$
$$\downarrow t = (t, \emptyset)$$
$$\downarrow \text{call } \ell = (\text{call } \ell, \emptyset)$$
$$\downarrow \mu t.L = (\mu t.L, \emptyset)$$

$$(r_1 \rightarrow r_2 : \{l_j(S_j), L_j\}_{j \in J}) \downarrow r = \begin{cases} (L_j, \emptyset) & \text{if } r_2 = r \text{ and } S_j = \text{unit and } J = \{j\} \\ \text{call } \ell, \ell = r_1 \rightarrow r_2 : \{l_j(S_j), L_j\}_{j \in J} & \text{if } r_2 = r \text{ and } S_j \neq \text{unit or } J \neq \{j\} \\ \text{where } \ell \text{ is a fresh name} & \text{where } L_j \downarrow r = (L'_j, D_j) \text{ for } j \in J. \end{cases}$$

We remark that the first case of the definition of $\downarrow$ for a branching type allows this function to eliminate a redundant interaction. Therefore one application of function $\uparrow L$ can get rid of more than one redundant interaction, as exemplified below.

The function $\uparrow L$ is defined by induction on the context in which the redundant interaction appears.

Definition 3 (The function $\uparrow L$). The application of the function $\uparrow L$ to $C[r_1 \rightarrow r_2 : l().L]$ for eliminating the interaction $r_1 \rightarrow r_2 : l().L$ is defined by induction on $C$:

$$\uparrow L(r_1 \rightarrow r_2 : l().L) = (L, \emptyset)$$

$$\downarrow r_1' \rightarrow r_2' : \{l_j(S_j), L_j, l'(S), C[r_1 \rightarrow r_2 : l().L]\}_{j \in J} =$$

$$\begin{cases} (r_1' \rightarrow r_2' : \{l_j(S_j), L_j, l'(S), C[L]\}_{j \in J}, \emptyset) & \text{if } r_2 = r_1' \text{ or } r_2 = r_2', \\ (r_1' \rightarrow r_2' : \{l_j(S_j), L'_j, l'(S), L'_j\}_{j \in J}, \bigcup_{j \in J} D_j \cup D) & \text{if } r_2 \neq r_1' \text{ and } r_2 \neq r_2', \\ & \text{where } L_j \downarrow r_2 = (L'_j, D_j) \text{ for } j \in J \\ & \text{and } \uparrow L(C[r_1 \rightarrow r_2 : l().L]) = (L', D) \end{cases}$$

The branching case of the above definitions needs some comments. If the receiver $r_2$ is also the sender or the receiver of the top branching, then she is aware of the choice of the label $l'$ and the redundant interaction can simply be erased. Otherwise $r_2$ must receive a communication in all branches $l_j$ and in this case these communications need to be replaced by calls to fresh names of light global types. For this reason we compute $L_j \downarrow r_2$ for all $j \in J$. Moreover we recursively call the mapping $\uparrow L$ on $C[r_1 \rightarrow r_2 : l().L]$.

We now show two applications of the lightening function, first to the global type $G$ of Figure 1 and then to the light global type obtained as a result of the first application. The application of $\uparrow L$ for eliminating the interaction $\text{req} \rightarrow \text{store} : \text{no4}()$ gives as a result $(L'_e, \ell_c = L_c)$, where $L_c$ is the type associated to $\ell_c$ in Figure 1 and:

$$L'_e = \text{req} \rightarrow \text{map} : \text{req1(str).map} \rightarrow \text{req} : \{\text{yes1(str).req} \rightarrow \text{issuer : req2(str).issuer} \rightarrow \text{req} : \{\text{yes2(int).call } \ell_c, \text{no2()}.end\}, \\ \text{no1()}.req \rightarrow \text{issuer : no5()}.end\}$$
Notice that also the redundant interaction \( \text{req} \rightarrow \text{store} : \text{no6()} \) is erased in \( L' \). The same result can be obtained by applying \( L \) for eliminating the interaction \( \text{req} \rightarrow \text{store} : \text{no6()} \). Now if we apply \( L \) to \( L'_a \) for eliminating \( \text{req} \rightarrow \text{issuer} : \text{no5()} \) we get \( (L_a, \ell_b = L_b) \), where \( L_a \) and \( L_b \) are the types associated to \( \ell_a \) and \( \ell_b \) in Figure[1] So we get all declarations shown in Figure[1]

4 Safety of the lightening function. In order to discuss the correctness of our lightening function, following [10] we view light global types with relative sets of declarations as denoting languages of interactions which can occur in multi-party sessions. The only difference is that recursive types do not reduce, the reasons being that our lightening function does not modify them. More formally the following definition gives a labelled transition system for light global types with respect to a fixed set of declarations. As usual \( \tau \) means a silent move, and \( \checkmark \) session termination.

**Definition 4 (LTS).**

\[
\begin{align*}
\text{call} \ell & \rightarrow_D L \\
\text{call} \ell & \rightarrow_D L' \\
\mu t. L & \rightarrow_D r \rightarrow r_2 : l().L & \rightarrow_D L & \text{end} \\
\text{act} & \\
J \neq \{ j \} \text{ or } S_j & \neq \text{unit} \\
r_1 \rightarrow r_2 & : \{ j(S_j), L_j \} \in J \\
& \rightarrow_d^{\ell_1 \rightarrow \ell_2, l(S_j)} D_j \\
\end{align*}
\]

We convene that \( \lambda \) ranges over \( r_1 \rightarrow r_2 : l(S) \) and \( \checkmark \), and that \( \sigma \) ranges over sequences of \( \lambda \). Using the standard notation:

\[
\begin{align*}
L & \rightarrow_D L' \quad \text{if} \quad L \rightarrow_D L_1 \rightarrow_D L_2 \rightarrow_D L' \quad \text{for some} \quad L_1, L_2 \\
L & \rightarrow_D L' \quad \text{if} \quad L \rightarrow_D L_1 \equiv_D L' \quad \text{for some} \quad L_1 \\
L & \equiv_D L \quad \text{if} \quad L \equiv_D L' \quad \text{for some} \quad L_1 \\
\end{align*}
\]

The language generated by \( L \) and relative to \( D \) is the set of sequences \( \sigma \) obtained by reducing \( L \) using \( D \). We take this language as the meaning of \( L \) relative to \( D \) (notation \( [L]_D \)).

**Definition 5 (Semantics).** \( [L]_D = \{ \sigma \mid L \equiv_D L' \text{ for some } L' \text{ or } L \equiv_D \} \).

Soundness of lightening then amounts to show that the function \( \downarrow \) preserves the meaning of light global types with respect to the relative sets of declarations. A first lemma shows soundness of the function \( \downarrow \).

**Lemma 1.** Let \( L \) be a light global type with declaration \( D \). If \( L \downarrow r = (L', D') \), then \( [L]_D = [L']_{D' \cup D} \).

**Proof.** The proof is by induction on \( L \) and by cases on Definition 2. The only interesting case is \( L = r_1 \rightarrow r_2 : \{ j(S_j), L_j \} \in J \).

1. If \( L \downarrow r = (L_j, \emptyset) \), then \( S_j = \text{unit} \) and \( J = \{ j \} \). By rule \([\text{RED}]\) we have \( L \rightarrow_D L_j \), thus \( [L]_D = [L_j]_D = [L']_D \).

2. If \( L \downarrow r = (\text{call} \ell, \ell = r_1 \rightarrow r_2 : \{ j(S_j), L_j \} \in J) \), then by rule \([\text{CALL}]\) we have \( L' \rightarrow_D (l = \ell) \cup D \). Thus \( [L]_D = [L']_{(l = \ell) \cup D} \).
3. If $L \downarrow x = (x_1 \rightarrow x_2 : \{l_j(S_j).L_j' \downarrow x\}_{j \in J}, \cup_{j \in J} D_j)$, then $L_j \downarrow x = (L_j', D_j')$ for $j \in J$. By Definition 5 and rule [ACT] $[L]_D = \cup_{j \in J} \{ x_1 \rightarrow x_2 : l_j(S_j) \cdot \sigma | \sigma \in [L_j]_D \}$ and $[L']_{\cup_{j \in J} D_j \cup D} = \cup_{j \in J} \{ x_1 \rightarrow x_2 : l_j(S_j) \cdot \sigma | \sigma \in [L_j']_{\cup_{j \in J} D_j \cup D} \}$. Since by induction $[L_j]_D = [L_j']_{D_j \cup D}$, we conclude $[L]_D = [L']_{D \cup D}$.

\[\square\]

**Theorem 1** (Soundness). Let $L$ be a light global type with set of declarations $D$. If $L(L) = (L', D')$, then $[L]_D = [L']_{D \cup D}$.

**Proof.** The proof is by induction on $L$ and by cases on Definition 3.

1. If $L = x_1 \rightarrow x_2 : (l().L''$, then $L(L) = (L'', \emptyset)$. We conclude since by Definition 5 and by rule [RED] $[L]_D = [L'']_D$.

2. Let $L = C_0[x_1 \rightarrow x_2 : l().L'']$ where $C_0 = x'_1 \rightarrow x'_2 : \{l_j(S_j).L_j,l'(S).C\}_{j \in J}$. By Definition 5 and by rule [ACT]

$[L]_D = \cup_{j \in J} \{ x'_1 \rightarrow x'_2 : l_j(S_j) \cdot \sigma | \sigma \in [L_j]_D \} \cup \{ x'_1 \rightarrow x'_2 : l'(S) \cdot \sigma | \sigma \in [C[x_1 \rightarrow x_2 : l().L'']]_D \}

(a) If $x_2 = x'_1$ or $x_2 = x'_2$, then

$[L']_D = \cup_{j \in J} \{ x'_1 \rightarrow x'_2 : l_j(S_j) \cdot \sigma | \sigma \in [L_j]_D \} \cup \{ x'_1 \rightarrow x'_2 : l'(S) \cdot \sigma | \sigma \in [C[L']_D \}

We conclude since rule [RED] $[C[x_1 \rightarrow x_2 : l().L'']]_D = [C[L']_D]_D$.

(b) If $x_2 \neq x'_1$ and $x_2 \neq x'_2$, let us assume $L_j \downarrow x_2 = (L_j', D_j)$ for $j \in J$ and $L(C[x_1 \rightarrow x_2 : l().L]) = (L_0,D_0)$. Then

$[L']_{\cup_{j \in J} D_j \cup D_0 \cup D} = \cup_{j \in J} \{ x'_1 \rightarrow x'_2 : l_j(S_j) \cdot \sigma | \sigma \in [L_j']_{\cup_{j \in J} D_j \cup D_0 \cup D} \} \cup \{ x'_1 \rightarrow x'_2 : l'(S) \cdot \sigma | \sigma \in [L_0]_{\cup_{j \in J} D_j \cup D_0 \cup D} \}

We conclude since by Lemma $[L_j]_D = [L_j']_{D_j \cup D}$ and by induction

$[C[x_1 \rightarrow x_2 : l().L'']]_D = [L_0]_{D_0 \cup D}$.

In the running example we get $[G]_0 = [L_0]_{L_0 = L_0} = [L_0]_{L_0 = L_0}$.

\[\square\]

5. **Related works and conclusion.** In the recent literature global types have been enriched in various directions by making them more expressive through roles [15] or logical assertions [5] or monitoring [4], more safe through security levels for data and participants [8][7] or reputation systems [6].

The more related papers are [14] and [9]. Demangeon and Honda introduce nesting of protocols, that is, the possibility to define a subprotocol independently of its parent protocol, which calls the subprotocol explicitly. Through a call, arguments can be passed, such as values, roles and other protocols, allowing higher-order description. Therefore global types in [14] are much more expressive than our light types. Carbone and Montesi [9] propose to merge together protocols interleaved in the same choreography into a single global type, removing costly invitations. Their approach is opposite to ours, and they deal with implementations, while we deal with types.

In this paper we show how to decompose interactions among multiple participants in order to remove redundant interactions, by preserving the meaning of (light) global types. We plan to implement our lightening function in order to experiment its practical utility in different scenarios.
Acknowledgements
We are grateful to Mariangiola Dezani-Ciancaglini for her valuable comments and feedbacks. We also thank PLACES reviewers for careful reading, since we deeply revised this article following their suggestions.

REFERENCES.

[1] Martin Berger & Kohei Honda (2003): The Two-Phase Commitment Protocol in an Extended π-Calculus. Available at http://dx.doi.org/10.1016/S1571-0661(05)82502-2 EXPRESS’00, 7th International Workshop on Expressiveness in Concurrency.

[2] Jan A. Bergstra, Alban Ponse & Scott A. Smolka: Handbook of Process Algebra.

[3] Lorenzo Bettini, Mario Coppo, Loris DAntoni, Marco De Luca, Mariangiola Dezani-Ciancaglini & Nobuko Yoshida (2008): Global Progress in Dynamically Interleaved Multiparty Sessions. In: CONCUR, LNCS 5201, Springer, pp. 418–433. Available at http://dx.doi.org/10.1007/978-3-540-85361-9_33

[4] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda & Nobuko Yoshida (2013): Monitoring Networks through Multiparty Session Types. In: FMOODS/FORTE, LNCS 7892, Springer, pp. 50–65. Available at http://dx.doi.org/10.1007/978-3-642-38592-6_5

[5] Laura Bocchi, Kohei Honda, Emilio Tuosto & Nobuko Yoshida (2010): A Theory of Design-by-contract for Distributed Multiparty Interactions. In: CONCUR, LNCS, Springer, pp. 162–176. Available at http://dx.doi.org/10.1007/978-3-642-15375-4_12

[6] Viviana Bono, Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2012): A Reputation System for Multirole Sessions. In: TGC, LNCS 7173, Springer, pp. 1–24. Available at http://dx.doi.org/10.1007/978-3-642-30065-3_1

[7] Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2011): Information Flow Safety in Multiparty Sessions. In: EXPRESS, EPTCS 64, pp. 16–31. Available at http://dx.doi.org/10.4204/EPTCS.64.2

[8] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Tamara Rezk (2010): Session Types for Access and Information Flow Control. In: CONCUR, LNCS 6269, Springer, pp. 237–252. Available at http://dx.doi.org/10.1007/978-3-642-15375-4_17

[9] Marco Carbone & Fabrizio Montesi (2012): Merging Multiparty Protocols in Multiparty Choreographies. In: PLACES, EPTCS 109, pp. 21–27. Available at http://dx.doi.org/10.4204/EPTCS.109.4

[10] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): On Global Types and Multiparty Sessions. Log. Meth. Comp. Scie. 8, pp. 1–45. Available at http://dx.doi.org/10.2168/LMCS-8(1:24)2012

[11] Tzu-Chun Chen (2013): Theories for Session-based Governance for Large-Scale Distributed Systems. Ph.D. thesis, Queen Mary, University of London, UK. Available at http://www.di.unito.it/~lambda/biblio/entry-TC2013.html

[12] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Denielou, Kohei Honda & Nobuko Yoshida (2011): Asynchronous Distributed Monitoring for Multiparty Session Enforcement. In: TGC, LNCS 7454, pp. 25–45. Available at http://dx.doi.org/10.1007/978-3-642-30065-3_2

[13] Tzu-Chun Chen & Kohei Honda (2012): Specifying Stateful Asynchronous Properties for Distributed Programs. In: CONCUR, LNCS 7454, pp. 209–224. Available at http://dx.doi.org/10.1007/978-3-642-32940-1_16

[14] Romain Demangeon & Kohei Honda (2012): Nested Protocols in Session Types. In: CONCUR, LNCS 7454, Springer, pp. 272–286. Available at http://dx.doi.org/10.1007/978-3-642-32940-1_20

[15] Pierre-Malo Denielou & Nobuko Yoshida (2011): Dynamic Multirole Session Types. In: POPL, ACM, pp. 435–446. Available at http://doi.acm.org/10.1145/1926385.1926435

[16] Yliès Falcone (2010): You Should Better Enforce Than Verify. In: Runtime Verification, LNCS, Springer, pp. 89–105. Available at http://dx.doi.org/10.1007/978-3-642-16612-9_9
[17] Jifeng He, M. B. Josephs & C. A. R. Hoare (1990): *A Theory of Synchrony and Asynchrony*. Available at [http://www.cs.ox.ac.uk/publications/publication8131-abstract.html](http://www.cs.ox.ac.uk/publications/publication8131-abstract.html). LaTeXed manuscript.

[18] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Denilou & Nobuko Yoshida (2012): *Structuring Communication with Session Types*. In: *To appear in COB 2012*. Available at [http://mrg.doc.ic.ac.uk/publications/structuring-communication-with-session-types/main.pdf](http://mrg.doc.ic.ac.uk/publications/structuring-communication-with-session-types/main.pdf).

[19] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen & Nobuko Yoshida (2011): *Scribbling Interactions with a Formal Foundation*. In: *ICDCIT*, LNCS 6536, pp. 55–75. Available at [http://dx.doi.org/10.1007/978-3-642-19056-8_4](http://dx.doi.org/10.1007/978-3-642-19056-8_4).

[20] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In: *ECOOP’91*, LNCS 512, pp. 133–147. Available at [http://dx.doi.org/10.1007/BFb0057019](http://dx.doi.org/10.1007/BFb0057019).

[21] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *POPL*, ACM, pp. 273–284. Available at [http://doi.acm.org/10.1145/1328438.1328472](http://doi.acm.org/10.1145/1328438.1328472).

[22] Dimitrios Kouzapas, Nobuko Yoshida & Kohei Honda (2011): *On Asynchronous Session Semantics*. In: *FMOODS/FORTE*, pp. 228–243. Available at [http://dx.doi.org/10.1007/978-3-642-21461-5_15](http://dx.doi.org/10.1007/978-3-642-21461-5_15).

[23] Ocean Observatories Initiative. [http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/](http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/).

[24] Olaf Owe, Martin Steffen & Arild B. Torjusen (2010): *Model Testing Asynchronously Communicating Objects using Modulo AC Rewriting*. *ENCS* 264(3), pp. 69–84. Available at [http://dx.doi.org/10.1016/j.entcs.2010.12.015](http://dx.doi.org/10.1016/j.entcs.2010.12.015).

[25] Scribble Project homepage. [www.scribble.org](http://www.scribble.org).

[26] Nobuko Yoshida, Raymond Hu, Rumyana Neykova & Nicholas Ng (2014): *The Scribble Protocol Language*. In: *TGC 2013*. Available at [http://dx.doi.org/10.1007/978-3-319-05119-2_3](http://dx.doi.org/10.1007/978-3-319-05119-2_3).