Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 171 (2007) 127–151
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Calculus of Global Interaction based on Session Types

Marco Carbone[1] [a] Kohei Honda[2] [b] Nobuko Yoshida[3] [a]

[a] *Department of Computing*
Imperial College London
London, United Kingdom

[b] *Department of Computer Science*
Queen Mary College
University of London
London, United Kingdom

Abstract

This paper proposes a calculus for describing *communication-centred programs* and discusses its use through a formal description of several use cases from real business protocols. The formalism, called *global calculus*, aims at representing global message flows as *structured communications*. The global calculus originates from the Choreography Description Language (CDL), a web service description language developed by W3C's WS-CDL Working Group. Its type discipline is based on *session types* which have been studied over long years in the context of the π-calculus [15,10,22,6]. Session types offer a high-level abstraction and articulation for complex communication behaviours, and play a fundamental role to guide the programmer towards a clear, well-structured description of business protocols.

Keywords: Web Services, Communication-Centred Programming, π-Calculus, Session Types.

1 Introduction

The purpose of the present paper is to introduce a communication-based calculus which fundamentally differs from existing concurrency formalisms. This calculus was born as a result of a dialogue with a W3C's working group on web service standards, and is based on the idea of global description of interactions. As its origin in W3C standardisation suggests, this formalism is the distillation of engineering needs for describing complex interaction which may occur in real world business processes. The associated long version [9] presents extensive examples of

[1] Email: carbonem@doc.ic.ac.uk
[2] Email: kohei@dcs.qmul.ac.uk
[3] Email: yoshida@doc.ic.ac.uk

doi:10.1016/j.entcs.2006.12.041

business protocols written in the proposed global formalism, together with a formal correspondence with a standard process formalism, the π-calculus.

The direct engineering background of the present work is *Web Services Choreography Description Language*, often abbreviated as WS-CDL [23], an XML-based web service description language developed by W3C's WS-CDL Working Group, in collaboration with invited scientists including the present authors. WS-CDL has been developed in order to meet the engineering needs for the development of business protocols on the world-wide web. The central engineering idea of WS-CDL is embodied by the term *choreography* in its name. The underlying intuition can be summarised as:

"*Dancers dance following a global scenario without a single point of control*".

WS-CDL is about writing down such a "global scenario": in computational terms, it is a language for describing business protocols from a global viewpoint such that the description can be executed by communications among distributed processes without a single point of control. In other words, if a designer writes a global description in WS-CDL, it should be realisable as communicating processes without any central controlling agent (which contrasts with the notion of *orchestration*, where one master component, "conductor", directly controls activity of one or more slave components). Thus the notion of choreography intrinsically *demands* an implementation framework of global protocols to be correctly realised in each location. For this purpose, types play a fundamental role in WS-CDL.

A broader background of the present work is the explosive growth of the Internet and world-wide. This engineering background makes it feasible and advantageous to develop applications which will be engaged in complex sequences of interactions among two or more parties. Another background is maturing of theories of processes centring on the π-calculus and its types. The introduced formalism is based on a notion of structured communication, called *session*. A session binds a series of communications between two parties into one, distinguishing them from communications belonging to other sessions. This is a standard practice in business protocols (where an instance of a protocol should be distinguished from another instance of the same or other protocols) and in distributed programming (where two interacting parties use multiple TCP connections for performing a unit of conversation). The type disciplines for sessions have been studied for many years in the context of the π-calculus [15,10,22,6], where it has been shown that they offer a high-level abstraction for communication behaviour upon which further refined reasoning techniques can be built. Not only sessions offer a natural articulation for description of global description of complex interaction behaviour but also session types play an essential role in the presented theory for guiding the programmer towards a correct implementation of the business protocols.

This paper introduces the formal calculus and the idea of session types with many illustrative examples to those without background of process calculi and their typing systems. Section 2 and 3 introduce and illustrate the global formalism (a distilled version of WS-CDL). Section 4 illustrates the session types for the calculus. Finally Section 5 demonstrates its expressiveness by showing a large example inspired from a

use case of WS-CDL, together with its implementation as communicating processes. Further examples and the full technical development of a theoretical basis of the formalism can be found in [9].

Related Work.

Global methods for describing communication have been practiced in different engineering scenes in addition to WS-CDL. Representative examples include Message Sequence Charts (MSC) [14,1] and UML diagrams. These notations offer a useful aid at the design/specification stage, but are not intended as full-fledged programming languages, lacking in e.g. general control structures and/or notations for value/state manipulation. Petri-nets [21] can also be seen as offering a global description, though they are more useful as a specification/analytical tool.

DiCons is a notation for global programming of Internet applications [2]. DiCons chooses to use programming primitives close to user's experience, such as web server invocation, email, and web form filing, rather than general communication primitives. Its semantics is given by either MSCs or direct operational semantics. DiCons differs from our global calculus in that it does not use communication-oriented types and that it is intended for a specific application domain rather than being presented as a generic calculus.

The present work shares with many recent works in its use of types for well-structured communication-centred programming [19,3]. In particular, our work is based on the studies on session type disciplines [20,15,11,10,22,6], and extends them to both global descriptions and intra-session parallel communications. All previous work are based on end-point languages and calculi.

Fournet, Gordon, Bhargavan and Corin studied security-related aspects of web services. In their recent work [5], the authors have implemented part of WS-Security libraries using a dialect of ML, and have shown how annotated application-level usage of these security libraries can be analysed with respect to their security properties by translation into the security-enhanced π-calculus.

Laneve and Padovani [16] give a model of orchestrations of web services using an extensions of π-calculus to join patterns. They propose a typing system for guaranteeing a notion of smoothness i.e. a constraint on input join patterns such that their subjects (channels) are co-located in order to avoid a classical global consensus problem during communication.

A bisimulation-based correspondence between choreography and orchestration in the context of web services has been studied by Busi et al. [8], where a notion of state variables is used in the semantics of the orchestration model. They operationally relate choreographies to orchestration. In [12], the same authors introduce SOCK, a calculus for web services based on end-point descriptions. This formalism models networks of participants and their local behaviour enhanced with a mechanism for controlling communication at run-time based on logical conditions on the participant's store. In our work, communication structures are abstracted as session types. Using session types, the error freedom can be statically guaranteed by a type inference algorithm. Their dynamic control allows concrete reasoning on

the communicated values, but may require heavy run-time mechanisms. It is an interesting topic to integrate these two methods for efficient and flexible control of interactions.

As we noted, global notations are often used for representing security protocols, for which there are several recent works which offer formalisations. Briais and Nestmann [7] present a global notation for representing protocol narrations and relate it to the π-calculus. Since their sole focus is on cryptographic protocols, their global formalism is not intended as a fully expressive language for describing communication-centred software behaviour. Concretely their formalism does not include such notions as channel-based communication, conditional, branching/selection, loops, as well as type disciplines for communication. Strand Space [13] is a mathematical structure for analysing properties of cryptographic protocols. It models cryptographic protocols as causal chains of interactions, and is often presented in a global notation similar to UML sequence diagrams. Strand space does not by itself offer a description language for communication-centred programs, but rather a mathematical structure for representing them as abstract events with causality and composability. Their methods for security analysis may however be applicable to our global calculus.

2 The Global Calculus

2.1 Basic Ideas

In this section and the next, we introduce the syntax, dynamic semantics and static semantics of the global calculus. The *syntax* defines the set of terms where each term gives a description of interaction among two or more participants.

The description of interactions in the global calculus centres on the notion of *session*, in which two interacting parties establish a private connection for engaging a series of interactions, possibly interleaved with other sessions. More concretely, processes first exchange fresh session channels for a newly created session, then use them for interactions belonging to the session (this is equivalent to the more implicit framework where identity tokens in message content are used for signifying a session). For example, a simple protocol between Seller and Buyer can be represented as a term thus: called *interaction*:

$$\begin{aligned} &\mathsf{Buyer} \to \mathsf{Seller} : \mathit{serv}(\boldsymbol{\nu}\mathsf{s})\,. \\ &\mathsf{Buyer} \to \mathsf{Seller} : s\langle \mathsf{RequestQuote}, \mathit{productName}, x\rangle\,. \\ &\mathsf{Seller} \to \mathsf{Buyer} : s\langle \mathsf{ReplyQuote}, \mathit{productPrice}, y\rangle.0 \end{aligned} \tag{1}$$

In (1) above, a Buyer asks for a service *serv* creating the new session channel s which will be used for in-session message exchange. Therefore, Buyer requests a quote for a product, specifying the product name and, through the same channel,

a Seller replies with the quote value. In a single interaction:

$$\mathsf{Buyer} \rightarrow \mathsf{Seller} : s\langle \mathsf{RequestQuote}, \mathit{productName}, x\rangle$$

Here Buyer specifies the sender, Seller the receiver, s a session identifier, RequestQuote the operation name, *productName* the value to be sent, and x the variable at the receiver's side which gets populated once the sending of a value is done.

Once terms are given, the *dynamic semantics* specifies an abstract notion of "computation" underlying the global calculus. This is represented as a transition from a global description to another global description, carrying out each step of interaction (e.g. exchange of a message). Since each participant may own its own local variables, such transition can also involve collection of local variables of the participants involved. The dynamic semantics is defined using an intuitive notation,

$$(\sigma, I) \;\rightarrow\; (\sigma', I')$$

which says a global description I in a state σ (which is the collection of all local states of the participants) will be changed into I' and a new configuration σ' results. This idea comes from the small-step semantics for imperative languages.

The static semantics of the global calculus assigns types to well-formed terms. Since the calculus is based on session, we use a type discpline where we represent a structured sequence of interactions between two parties in a session as an type. Here "types" mean syntactic annotation on descriptions of interactions: in the present case, this annotation describes an abstract notion of interface of a service (or a shared service channel), and is inferred by typing rules for each description following its syntactic structure. For example, taking the example 1 above, the interactions at s can be abstracted by the following session type:

$$s \uparrow \mathsf{RequestQuote}(\mathtt{String}).\ s \downarrow \mathsf{ReplyQuote}(\mathtt{Int}) \tag{2}$$

The session type in (2) abstracts a sequence of actions performed at ch, specifying the following abstract behaviour:

> *First send ("↑") a string with operation name* RequestQuote*, then receive ("↓") an integer with operation name* ReplyQuote*.*

Note this abstraction is given from the Buyer's viewpoint. Similarly, we can present the abstraction from the Seller's viewpoint:

$$s \downarrow \mathsf{RequestQuote}(\mathtt{String}).\ s \uparrow \mathsf{ReplyQuote}(\mathtt{Int}) \tag{3}$$

which simply reverses the direction of information flows. Note that, in this way, there is a natural notion of **duality** associated with session types [15].

2.2 Formal Syntax

The formal syntax of the global calculus is given by standard BNF. Below symbols $I, I', \ldots$ denote *terms* of the global calculus, also called *interactions*. Terms describe a course of information exchange among two ore more parties from a global viewpoint.

$$
\begin{array}{rll}
I ::= & A \to B : ch(\boldsymbol{\nu}\, \tilde{s})\,.\, I & \text{(init)} \\
| & A \to B : s\langle op, e, y\rangle\,.\, I & \text{(com)} \\
| & x@A := e\,.\, I & \text{(assign)} \\
| & \mathsf{if}\ e@A\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2 & \text{(ifthenelse)} \\
| & I_1 + I_2 & \text{(sum)} \\
| & I_1 \mid I_2 & \text{(par)} \\
| & (\boldsymbol{\nu} s) I & \text{(new)} \\
| & X & \text{(recVar)} \\
| & \mathbf{rec}\ X\,.\, I & \text{(rec)} \\
| & \mathbf{0} & \text{(inaction)}
\end{array}
$$

The grammar above uses the following symbols[4]:

- $a, b, c, ch, \ldots$ range over a collection Ch of *service channels* (also called *session initiating channels*). They may be considered as shared channels of web services.
- $s, s', \ldots$ range over a collection $\mathcal{S}$ of *session channels*. Session channels designate communication channels freshly generated for each session. They can be implemented in various ways: in TCP, the same concept is realised by so-called *connection* (also called *session*). In web services, they are realised by sending freshly generated identity information as part of messages.
- $A, B, C, \ldots$ range over a collection $\mathcal{P}$ of *participants*. Participants are those who are engaged in communications with others, each equipped with its own local state. Each participant may have more than one threads of interactions using multiple channels.
- $x, y, z, \ldots$ range over a collection *Var* of *variables*, which are close to variables in the traditional programming languages such as Pascal and C, in that their content is updatable.
- $X, Y, Z, \ldots$ range over a collection of *term variables*, which are used for representing recurrence (loop) in combination with recursion **rec** $X.I$. Note term variables occur annotated with participants.
- $e, e', \ldots$ range over *expressions*, given by the grammar $e ::= x \mid v \mid f(e_1, \ldots, e_k)$ where f ranges over an appropriate set of function symbols (including standard arithmetic/boolean operators) and $v, w, \ldots$ range over atomic values such as natural numbers and booleans.

[4] As standard, we assume there is an unbounded supply of distinct symbols in each syntactic category.

Each construct in the above grammar is illustrated in the next subsection. We write $A \to B : s\langle \mathsf{op} \rangle \,.\, I$ whenever the expression e and the variable x are not important in $A \to B : s\langle op, e, x \rangle \,.\, I$.

2.3 Illustration of Syntax

The initial two constructs represent communications. First, $A \to B : b(\boldsymbol{\nu}\, \tilde{s}) \,.\, I$ indicates that A invokes a service channel b at B and initiates a new session that will use fresh session channels $\tilde{s}$, followed by interaction I. Subsequent communications in I belonging to this session are done through $\tilde{s}$ (I can have other communications belonging to different sessions). In the main part of the present study we shall assume A and B are distinct [5]. As $\tilde{s}$ should be local to the session (i.e. unknown outside), we set each $s_i \in \tilde{s}$ to be bound in I.

Second, $A \to B : s\langle op, e, y \rangle$ expresses the sending action by A whose message consists of a selected operator op, with the receiver B. The value of the expression e (which can only use variables located at A) is assigned to the variable y (which must be located at B).

Third, another primitive operation is the *assignment* $x@A := e \,.\, I$. The operation is local at the specified participant (A above), where variable x at A is updated with the result of evaluating e, also located at A.

We can use the conditional $\mathsf{if}\ e@A\ \mathsf{then}\ I_1\ \mathsf{else}\ I_2$ or the summation $I_1 + I_2$ to branch the course of actions. The first operation evaluates e once and, if it evaluates to true, the branch I_1 will be executed, else branch I_2. Note the condition e is located at A. The second operation implicitly selects one of the branches by non-deterministically behaving either as I_1 or as I_2. The summation operator $+$ is commutative and associative, so that we often write $\Sigma_i I_i$ for the n-fold sum of interactions.

The term $I_1 \mid I_2$ denotes the parallel composition. However, unlike the standard process calculi, there is no communication between I_1 and I_2: $I_1|I_2$ just means two independent threads of interactions.

$(\boldsymbol{\nu} s)I$ denotes restriction (or hiding) of a session channel, where $(\boldsymbol{\nu} s)$ binds free occurrences of s in I. This is used for designating newly created session channels when a session is initiated. The term $(\boldsymbol{\nu}\tilde{s})I$ stands for a sequence of restrictions. Since restriction is only added when an outermost initialisation prefix reduces, it is natural to assume that *restrictions never occur under prefixes initiation, communication and assignment nor do they occur in a summand of a summation.*

Interaction which can be repeated unboundedly is realised by recursion. Then the term $\mathbf{rec}\ X \,.\, I$ is the standard recursion construct, where $\mathbf{rec}\ X$ is called *recursor*, with X, the recursion variable, binding its free occurrences in I. Finally, $\mathbf{0}$ is the inaction, representing the lack of actions.

[5] This is a natural constraint if we wish to describe inter-participants interactions, which are often the only focus of many business protocols. The theoretical framework of the present study works intact when we allow intra-participant interactions.

2.4 Examples

We illustrate the syntax through simple examples. These examples will be used throughout the paper as running examples.

Example 1 (Syntax, 1) Consider the following interaction:

$$\begin{array}{l} \{\mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle \,.\, I_1\} \\ \qquad\qquad + \\ \{\mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle \,.\, I_2\} \end{array} \tag{4}$$

In this example, Buyer and Seller are the participants involved, *B2Sch* is a session channel name and QuoteAccept and QuoteReject are operation names. Thus, as a whole, (4) can be read as follows:

> *Through a session channel* B2Sch*, Buyer selects one of the two options offered by Seller,* QuoteAccept *and* QuoteReject*. If the first option is selected, Buyer sends the quote "100" which will be stored in x by Seller and proceeds to I_1. In the other case, Seller sends the abort number stored in the variable x_{AbortNo} which will be stored in y by the Seller and proceeds to I_2.*

We interpret the sum + as an *internal sum* for Buyer (i.e. Buyer initiates this choice) and as an *external sum* for Seller (i.e. Seller passively waits for one of the branches (operators) to be chosen by the environment).

Example 2 (Syntax, 2) The following is a refinement of the description above:

$$\begin{array}{l} \textbf{if}\ \ x_{\mathtt{quote}} \leq 1000\,@\mathtt{Buyer}\ \ \textbf{then} \\ \quad \{\ \ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle \,.\, I_1 \quad \} \\ \textbf{else} \\ \quad \{\ \ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle \,.\, I_2 \ \ \} \end{array} \tag{5}$$

This description now specifies the "reason" why each branch is taken. Notice the condition in the conditional branch, $x \leq 1000$, is explicitly *located*: the description says this judgement takes place at Buyer. Note also the description is still the external choice for Seller: it is Buyer who selects one of the options, which Seller waits for passively. The description becomes self-contained by adding an initial

session invocation at a service channel, say ch, and a request for a quote.

$$
\begin{array}{l}
\mathsf{Buyer} \to \mathsf{Seller} : \mathsf{ch}(\boldsymbol{\nu}\ \mathit{B2Sch}, \mathit{S2Bch})\,\mathbf{.} \\
\mathsf{Seller} \to \mathsf{Buyer} : \mathit{S2Bch}\langle \mathsf{Quote}, \mathit{100}, y\rangle\,\mathbf{.} \\
\textbf{if}\ \ x_{\mathtt{quote}} \leq 1000\,@\mathtt{Buyer}\ \ \textbf{then} \\
\quad \{\ \ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle\,\mathbf{.}\, I_1 \quad \} \\
\textbf{else} \\
\quad \{\ \ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle\,\mathbf{.}\, I_2 \ \ \}
\end{array}
\tag{6}
$$

In here, Buyer invokes Seller to initiate a session with the session channels *B2Sch* and *S2Bch*.

Example 3 (Syntax, 3) A session can have multiple session names for communication. This is the standard practice in business protocols and other interaction-centred applications, and is essential to have multiple parallel interactions inside a single session. As an example, suppose that Buyer wants to start a session at a channel acc in which it communicates acceptance of a quote on a session name *Op* and, in parallel, sends its address on a session name *Data*. This can be expressed as:

$$
\begin{array}{l}
\mathsf{Buyer} \to \mathsf{Seller} : \mathsf{acc}(\boldsymbol{\nu}\ \mathit{Op}, \mathit{Data})\,\mathbf{.} \\
\quad \{\mathsf{Buyer} \to \mathsf{Seller} : \mathit{Op}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle\,\mathbf{.}\,\mathbf{0} \quad | \\
\qquad \mathsf{Buyer} \to \mathsf{Seller} : \mathit{Data}\langle \mathsf{QuoteAccept}, \mathit{Address}, y\rangle\,\mathbf{.}\,\mathbf{0} \quad \}
\end{array}
\tag{7}
$$

Here, two session channels, *Op* and *Data*, are communicated at the time of session initiation at channel acc.

Examples of other constructs, such as recursion and hiding, will be given in the last section.

3 Dynamics of the Global Calculus

3.1 Basic ideas of Reduction

From an engineering viewpoint, the dynamic semantics pins down a mathematical notion which designers, implementers and users can refer to when they wish to discuss about dynamic behaviour of description with rigour. For example, this would allow us to state with precision whether an implemented program conforms to the description or not. Another example usage is in monitoring, where a monitor would check the execution based on the stipulated formal dynamic semantics of a given description.

Computation in the global calculus is represented by a step-by-step transition, each step consisting of the execution of a primitive operation, such as communication, assignment or conditional, which has an effect on on the local state of an involved participant.

To formalise this idea, we use a *configuration* which is a pair of a *state* (a collection of the local states of all participants involved) and an interaction, written $(\sigma,\ I)$. Formally a *state*, ranged over by $\sigma, \sigma', \ldots$, is a function from $Var \times \mathcal{P}$ to Val, i.e. a variable at each participant is assigned a value in a store. We shall write $\sigma@A$ to denote the portion of σ local to A, and $\sigma[y@A \mapsto v]$ to denote a new state which is identical with σ except that $\sigma'(y, A)$ is equal to v. The dynamics is defined in the form:

$$(\sigma,\ I)\ \rightarrow\ (\sigma',\ I')$$

which says I in the configuration σ performs one-step computation (which can be assignment, interaction, etc.) and becomes I' with the new configuration σ'. The relation $\rightarrow$ is called *reduction* or *reduction relation*[6]. For example, communication action will change both the state and the term shape:

$$(\sigma,\ A \rightarrow B : s\langle \mathsf{send}, 3, x\rangle.I) \quad \rightarrow \quad (\sigma[x@B \mapsto 3],\ I)$$

which indicates:

> *"A sends a message* send *with value* 3*, which is received by B and stored in the variable x at B, with the residual interaction I".*

Note communication action happens automatically, without first having sending and receiving actions separately and then having their synchronisation. Assignment is treated similarly.

$$(\sigma,\ x@B := 3.I) \quad \rightarrow \quad (\sigma[x@B \mapsto 3],\ I)$$

Since an assignment is located, only x at B is updated, and the next interaction I is unfolded. Interaction can involve choice, where one of the branches is chosen nondeterministically, i.e. we can have either:

$$(\sigma,\ I_1 + I_2)\ \rightarrow\ (\sigma,\ I_1) \qquad \text{or} \qquad (\sigma,\ I_1 + I_2)\ \rightarrow\ (\sigma, I_2)$$

and they are both legitimate reductions.

The conditional depends on the evaluation of an expression. For example, assuming x at A stores 0, we have

$$(\sigma,\ \mathsf{if}\ x@A = 0\ \mathsf{then}\ A \rightarrow B : s\langle \mathsf{ok}, 3, x\rangle.I_1\ \mathsf{else}\ ...) \quad \rightarrow \quad (\sigma[x@B \mapsto 3],\ I_1)$$

But if x at A stores say 1, then the second branch will be selected.

For recursion, we expect a recurring behaviour. For example, the following behaviour just continues to assign 1 to a variable:

$$(\sigma,\ \mathbf{rec}\ X.x@B := 1.X)\ \rightarrow\ (\sigma[x@B \mapsto 3],\ \mathbf{rec}\ X.x@B := 1.X)$$

In the following subsection, we illustrate the notion of reduction for each construct one by one.

[6] The term "reduction" originally comes from the λ-calculus, where the sole purpose of computation is to reduce to a final answer of calculation. While it is not entirely suitable for interaction computation, we use the term from convention and from our respect to the basic formalism which started semantics studies.

3.2 Reduction Rules.

Reduction relation is defined by having one rule for each construct, together with associated rules. First we have a rule for session-initiating communication:

$$(\textsc{Init})\ \frac{-}{(\sigma,\ A \to B : b(\boldsymbol{\nu}\ \tilde{s})\,.\,I) \to (\sigma, (\boldsymbol{\nu}\tilde{s})I)}$$

where $\tilde{s}$ is a vector of one or more pairwise distinct session channels. The rule says that, after A communicates with B for session initiation with fresh session channels $\tilde{s}$, A and B share $\tilde{s}$ locally (indicated by ν-binding), and the next I is unfolded. The state σ stays as it is since no communication of values takes place.

We have already seen an example of reduction representing communication through a session channel. We now report the the formal rule:

$$(\textsc{Comm})\frac{\sigma \vdash e@A \Downarrow v}{(\sigma,\ A \to B : s\langle op, e, x\rangle\,.\,I) \to (\sigma[x@B \mapsto v],\ I)}$$

The premise of the rule above uses the judgement (called *evaluation judgement*):

$$\sigma \vdash e@A \Downarrow v$$

which says:

Expression e is evaluated into the value v in the A-portion of the state σ.

For example, if σ says x at A stores 3, then we have $\sigma \vdash (x + x)@A \Downarrow 6$. Thus the expression to be communicated is evaluated in the *source* part of the state: and the value communicated is assigned in the *target* part of the state.

The formal rule for assignment is given as:

$$(\textsc{Assign})\frac{\sigma \vdash e@A \Downarrow v}{(\sigma, x@A := e\,.\,I) \to (\sigma[x@A \mapsto v],\ I)}$$

which updates the state at the participant A and unfolds the next interaction.

The rules for conditional assumes, again using the evaluation judgement, that the conditional expression evaluates to either tt (for truth) or ff (for falsity). In the former:

$$(\textsc{IfTrue})\frac{\sigma \vdash e@A \Downarrow \mathsf{tt}}{(\sigma,\ \mathsf{if}\ e@A\ \mathsf{then}\ I\ \mathsf{else}\ {}_1I_2) \to (\sigma, I_1)}$$

Symmetrically, when the condition evaluates to the falsity:

$$(\textsc{IfFalse})\frac{\sigma \vdash e@A \Downarrow \mathsf{ff}}{(\sigma,\ \mathsf{if}\ e@A\ \mathsf{then}\ I\ \mathsf{else}\ {}_1I_2) \to (\sigma,\ I_2)}$$

The rule for summation is standard:

$$(\textsc{Sum})\frac{-}{(\sigma,\ I_1 + I_2) \to (\sigma',\ I_1)}$$

The symmetric rule is subsumed by structural congruence which will be introduced later on.

For parallel composition, the rule is defined just by considering interleaving of two components. Thus we define:

$$(\textsc{Par})\frac{(\sigma,\ I_1) \to (\sigma',\ I_1')}{(\sigma, I_1 \mid I_2) \to (\sigma',\ I_1' \mid I_2)}$$

where we reduce the left-hand side. The symmetric rule is again subsumed by the use of the structural rules we stipulate later.

For restriction we have:

$$(\textsc{Res})\frac{(\sigma,\ I) \to (\sigma',\ I')}{(\sigma, (\boldsymbol{\nu}\tilde{s})I) \to (\sigma',\ (\boldsymbol{\nu}\tilde{s})I')}$$

which says restriction does not affect reduction. For recursion, we use the standard unfolding rule.

$$(\textsc{Rec})\frac{(\sigma,\ I[\mathbf{rec}\ X.I/X]) \to (\sigma',\ I')}{(\sigma,\ \mathbf{rec}\ X.I) \to (\sigma',\ I')}$$

The rule says that:

> *If the unfolding of* **rec** $X.I$, $I[\mathbf{rec}\ X.I/X]$ *(which substitutes* **rec** $X.I$ *for each free* X *in* I*) under* σ *reduces to* I' *with the resulting state* σ'*, then* **rec** $X.I$ *itself under* σ *will reach* $(\sigma',\ I')$.

Finally the inaction $\mathbf{0}$ does not have any reduction. We also use the following rule, which says that when we reduce we take terms up to a certain equality, following [4,17].

$$(\textsc{Struct})\frac{I \equiv I'' \qquad (\sigma,\ I) \to (\sigma,\ I') \qquad I' \equiv I'''}{(\sigma,\ I'') \to (\sigma',\ I''')}$$

where the structural equality $\equiv$ is defined by the following rules:

- $I \equiv I'$ $(I \equiv_\alpha I'\ )$
- $I + I \equiv I,\ \ I_1 + I_2 \equiv I_2 + I_1,\ \ (I_1 + I_2) + I_3 \equiv I_1 + (I_2 + I_3)$,
- $I|\mathbf{0} \equiv I,\ \ I_1|I_2 \equiv I_2|I_1,\ \ (I_1|I_2)|I_3 \equiv I_1|(I_2|I_3)$,
  $((\boldsymbol{\nu}s)I_1)|I_2 \equiv (\boldsymbol{\nu}s)(I_1|I_2)$ with $s \notin \mathsf{fn}(I_2)$)

In the last rule, $\mathsf{fn}(I)$ denotes the free names (including variables, channels and session channels) occurring in I. The relation $\equiv$ is the least congruence on terms including the above equations. While the benefit of the use of structural rules in reduction rules is limited in the present context (in comparison with standard process calculi), considering terms up to $\equiv$ is often natural and adds clarity in practice.

3.3 Examples of Reduction.

Example 4 (Reduction: Communication) Recall the following term from Example 1:

$$I_0 \stackrel{\text{def}}{=} \{\mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle \,.\, I_1\} + \{\mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle \,.\, I_2\} \tag{8}$$

We infer the reductions of I_0. There is one reduction for each branch. For the first summand, we note $\sigma \vdash 100@\mathsf{Buyer} \Downarrow 100$ and infer, in two steps, using (Comm):

$$(\sigma,\ I_0) \to \ldots \to (\sigma[x@\mathsf{Seller} \mapsto 100],\ I_1) \tag{9}$$

Similarly we have the following reduction for the second branch. Assume x_{AbortNo} stores (say) 28 at Buyer in σ, hence $\sigma \vdash x_{\mathsf{AbortNo}}@\mathsf{Buyer} \Downarrow 28$,

$$(\sigma,\ I_0) \to (\sigma[y@\mathsf{Seller} \mapsto 28],\ I_2) \tag{10}$$

Example 5 (Reduction: Conditional) We now show reduction for Example 2. First we reproduce the term.

$$I_0' \stackrel{\text{def}}{=} \begin{array}{l} \textbf{if}\ \ x_{\mathtt{quote}} \leq 1000\,@\mathtt{Buyer}\ \ \textbf{then} \\ \quad \{\ \ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle \,.\, I_1' \ \ \} \\ \textbf{else} \\ \quad \{\ \ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle \,.\, I_2' \ \ \} \end{array} \tag{11}$$

If we assume $\sigma@\mathsf{Buyer}(x_{\mathtt{quote}}) = 800$ then we can infer by rule (IfTrue):

$$\frac{\sigma \vdash (800 \leq 1000)@\mathsf{Buyer} \Downarrow \mathsf{tt}}{(\sigma,\ I_0') \to (\sigma,\ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteAccept}, \mathit{100}, x\rangle \,.\, I_1')} \tag{12}$$

Further applying (Comm) to the resulting configuration, we conclude:

$$\begin{array}{rl} (\sigma,\ I_0') & \to (\sigma,\ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathit{QuoteAccept}, \mathit{100}, x\rangle \,.\, I_1') \\ & \to (\sigma[x@\mathsf{Seller} \mapsto 100],\ I_1') \end{array}$$

which is the only reduction sequences from $(\sigma,\ I_0')$ in this case. Assume on the other hand $\sigma@\mathsf{Buyer}(x_{\mathtt{quote}}) = 1200$. Then we have by rule (IfFalse):

$$\frac{\sigma \vdash (1200 \leq 1000)@\mathsf{Buyer} \Downarrow \mathsf{ff}}{(\sigma,\ I_0') \to (\sigma,\ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle \,.\, I_2')} \tag{13}$$

Hence in this case we have:

$$\begin{array}{rl} (\sigma,\ I_0') & \to (\sigma,\ \mathsf{Buyer} \to \mathsf{Seller} : \mathit{B2Sch}\langle \mathsf{QuoteReject}, x_{\mathsf{AbortNo}}, y\rangle \,.\, I_2') \\ & \to (\sigma[y@\mathsf{Seller} \mapsto 28],\ I_2) \end{array}$$

which is again the only possible reduction sequence under the assumption.

Example 6 (Reduction: Init, Par and Struct) We next consider Example 3:

$$\begin{array}{rcl} J_0 & = & \mathsf{Buyer} \to \mathsf{Seller} : \mathsf{acc}(\boldsymbol{\nu}\ Op, Data)\,. \\ & & \quad \{\mathsf{Buyer} \to \mathsf{Seller} : Op\langle \mathsf{QuoteAccept}, 100, x\rangle\,.\,\mathbf{0} \quad | \\ & & \quad\ \mathsf{Buyer} \to \mathsf{Seller} : Data\langle \mathsf{QuoteAccept}, w_{\mathsf{Address}}, y\rangle\,.\,\mathbf{0} \quad \} \end{array} \tag{14}$$

Let us denote the two components of the parallel composition with J_1 and J_2. Then, by (Init), we obtain:

$$(\sigma,\ J_0) \to (\sigma,\ (\boldsymbol{\nu} Op, Data)(J_1 \mid J_2)) \tag{15}$$

By (Comm) we have: $(\sigma,\ J_1) \to (\sigma[x@\mathsf{Seller} \mapsto 100],\ \mathbf{0})$, hence by (Par) we conclude with:

$$(\sigma,\ J_1|J_2) \to (\sigma[x@\mathsf{Seller} \mapsto 100],\ \mathbf{0}|J_2) \tag{16}$$

For the symmetric case, assume $\sigma@\mathsf{Buyer}(w_{\mathsf{Address}}) = adr$ (where adr is a string standing for an address) Then by (Comm) we have $(\sigma,\ J_2) \to (\sigma[y@\mathsf{Seller} \mapsto adr],\ \mathbf{0})$, hence by (Par) we arrive at:

$$(\sigma,\ J_2 \mid J_1) \to (\sigma[y@\mathsf{Seller} \mapsto adr],\ \mathbf{0} \mid J_1) \tag{17}$$

Noting $J_1 \mid J_2 \equiv J_2 \mid J_1$, we can now apply ((Struct)) to obtain:

$$(\sigma,\ J_1 \mid J_2) \to (\sigma[y@\mathsf{Seller} \mapsto adr],\ J_1) \tag{18}$$

Note we also simplified the resulting term. In summary, we have two sequences of reductions up to $\equiv$:

$$(\sigma,\ J_0) \ \to\ (\sigma,\ (\boldsymbol{\nu}\, Op, Data)(J_1|J_2)) \ \to\ (\sigma[x@\mathsf{Seller} \mapsto 100],\ (\boldsymbol{\nu} Data)J_2) \ \to\ (\sigma',\ \mathbf{0})$$

and

$$(\sigma,\ J_0) \ \to\ (\sigma,\ (\boldsymbol{\nu} Op, Data)(J_1|J_2)) \ \to\ (\sigma[y@\mathsf{Seller} \mapsto adr],\ (\boldsymbol{\nu} Op)J_1) \ \to\ (\sigma',\ \mathbf{0})$$

where we set $\sigma' \stackrel{\text{def}}{=} \sigma[x@\mathsf{Seller} \mapsto 100][y@\mathsf{Seller} \mapsto adr]$.

Example 7 (Reduction: Recursion) Finally we show an example of recursion reduction, taking the example $\mathbf{rec}\ X.(x@B := 1.X)$ before. Noting:

$$(x@B := 1.X)[\mathbf{rec}\ X.x@B := 1.X/X] \quad \stackrel{\text{def}}{=} \quad x@B := 1\,;\, \mathbf{rec}\ X.x@B := 1.X$$

hence we have:

$$\begin{array}{rcl} (\sigma,\ \mathbf{rec}\ X.x@B := 1.X) & \to & (\sigma[x@B \mapsto 1],\ \mathbf{rec}\ X.x@B := 1.X) \\ & \to & (\sigma[x@B \mapsto 1],\ \mathbf{rec}\ X.x@B := 1.X) \\ & \to & .... \end{array}$$

as expected.

4 Session Types for the Global Calculus

4.1 Syntax of Session Types

As briefly mentioned in §2.1, we use session types [15] as the type structures for the global calculus. In advanced web services and business protocols, the structures of interaction in which a service/participant is engaged in may not be restricted to one-way messages or RPC-like request-replies. This is why the type abstraction of the global calculus needs to capture a complex interaction structure of services, leading to the use of session types. By having structure of interactions explicitly as type abstraction, programmers are encouraged to think the communication-centred programs at a more abstract level than regarding interactions as a collection of unrelated communications, focussing on their high-level structures. The grammar of types follow.

$$\begin{array}{lcl} \theta & ::= & \texttt{bool} \mid \texttt{int} \mid \ldots \\ \alpha & := & \Sigma_i s \downarrow op_i(\theta_i)\,.\,\alpha_i \mid \Sigma_i s \uparrow op_i(\theta_i)\,.\,\alpha_i \mid \alpha_1 \mid \alpha_2 \mid \mathbf{t} \mid \mathbf{rec}\ \mathbf{t}.\alpha \mid \texttt{0} \end{array}$$

Above $\theta, \theta', \ldots$ range over *value types*, which in the present case only include atomic data types. $\alpha, \alpha', \ldots$ are *session types*. Note session channels $s, s', \ldots$ occur free in session types (this is necessary because of multiple session channels in a single session, cf. [15]). We take $\mid$ to be commutative and associative, with the identity $\texttt{0}$. Recursive types are regarded as regular trees in the standard way [18]. Brief illustration of each construct follows.

- $\Sigma_i s \downarrow op_i(\theta_i)\,.\,\alpha_i$ is a *branching input type at* s, indicating possibilities for receiving any of the operators from $\{op_i\}$ (which should be pairwise distinct) with a value of type θ_i.
- $\Sigma_i s \uparrow op_i(\theta_i)\,.\,\alpha_i$, a *branching output type at* s, is the exact dual of the above.
- $\alpha_1 \mid \alpha_2$ is a *parallel composition of* α_1 *and* α_2, abstracting parallel composition of two sessions. We demand session channels in α_1 and those in α_2 are disjoint.
- $\mathbf{t}$ is a *type variable*, while $\mathbf{rec}\ \mathbf{t}.\alpha$ is a *recursive type*, where $\mathbf{rec}\ \mathbf{t}$ binds free occurrences of $\mathbf{t}$ in α. A recursive type represents a session with a loop. We assume each recursion is guarded, i.e., in $\mathbf{rec}\ \mathbf{t}.\alpha$, the type α should be either an input/output type or n-ary parallel composition of input/output types.
- $\texttt{0}$ is the *inaction type*, indicating termination of a session. $\texttt{0}$ is often omitted.

Each time a session occurs at a shared service channel, session channels are freshly generated and exchanged. Thus the interface of a service should indicate a vector of session channels to be exchanged, in addition to how they are used. This is represented by *abstract session type*, or *service type*, in which concrete instances of session channels in a session type are abstracted, written: $(\tilde{s})\ \alpha$ where $\tilde{s}$ is a vector of pairwise distinct session channels which should cover all session channels in α, and α does not contain free type variables. $(\tilde{s})$ binds occurrences of session channels in $\tilde{s}$ in α, which induces the standard alpha-equality.

Before illustrating these types with examples, we introduce a natural notion of duality. The *co-type*, or *dual*, of α, written $\overline{\alpha}$, is given as follows.

$$\overline{\Sigma_i s_i \uparrow op_i(\theta_i) \,.\, \alpha_i} = \Sigma_i s_i \downarrow op_i(\theta_i) \,.\, \overline{\alpha_i} \qquad \overline{\Sigma_i s_i \downarrow op_i(\theta_i) \,.\, \overline{\alpha_i}} = \Sigma_i s_i \uparrow op_i(\theta_i) \,.\, \overline{\alpha_i}$$

$$\overline{\mathbf{rec}\ t \,.\, \alpha} = \mathbf{rec}\ t \,.\, \overline{\alpha} \qquad \overline{\mathbf{t}} = \mathbf{t} \qquad \overline{0} = 0$$

For example, the co-type of $s \downarrow \mathsf{QuoteReq}(\mathtt{string}).0$ is $s \uparrow \mathsf{QuoteReq}(\mathtt{string}).0$, exchanging input and output.

Given a term I, we type I with service/session types in the following way:

$$\Gamma \vdash I \triangleright \Delta$$

where Γ is a mapping from service channels to service types, and Δ is a mapping from session channels to session types. We call $\Gamma \vdash I \triangleright \Delta$, a *typing sequent*. A typing sequent $\Gamma \vdash I \triangleright \Delta$ says that the usage of service/session channels in I conforms to Γ and Δ, respectively. Such a sequent is derived using *typing rules*, which construct service/session typings for a term I following the latter's syntactic structure. We leave the presentation of typing rules to [9], in which we also show that they satisfy the standard properties such as subject reduction and principal typing. Here, in the next subsection, we rather illustrate typing through concrete examples.

4.2 Examples of Session Types

Example 8 (Session Type: basics) Consider the following interaction (cf. Example 1), assuming *adr* and *prd* are variables of `string` type, located at both Buyer and Seller.

$$\begin{aligned} &\mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{QuoteReq}, \mathit{prd}, \mathit{prd}\rangle. \\ &\mathsf{Seller} \to \mathsf{Buyer} : s_2\langle \mathsf{QuoteRep}, \mathit{100}, y\rangle. \\ &\mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Purchase}, \mathit{adr}, \mathit{adr}\rangle.\mathbf{0} \end{aligned} \tag{19}$$

The interface which Seller offers (as far as this interaction goes) can be described by the following session type:

$$s_1 \downarrow \mathsf{QuoteReq}(\mathtt{string}).\ s_2 \uparrow \mathsf{QuoteRep}(\mathtt{int}).\ s_1 \downarrow \mathsf{Purchase}(\mathtt{string}).\ 0 \tag{20}$$

the same interaction can be type-abstracted from the viewpoint of Buyer:

$$s_1 \uparrow \mathsf{QuoteReq}(\mathtt{string}).\ s_2 \downarrow \mathsf{QuoteRep}(\mathtt{int}).\ s_1 \uparrow \mathsf{Purchase}(\mathtt{string}).\ 0 \tag{21}$$

which is nothing but the co-type of (20). Now let us add a session initiation to (20):

$$\begin{array}{l} \mathsf{Buyer} \to \mathsf{Seller} : \mathsf{ch}(\mathsf{s}_1\mathsf{s}_2). \\ \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{QuoteReq}, prd, prd\rangle. \\ \mathsf{Seller} \to \mathsf{Buyer} : s_2\langle \mathsf{QuoteRep}, 100, y\rangle. \\ \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{QuoteAcc}, adr, adr\rangle.\mathbf{0} \end{array} \tag{22}$$

Then the service type of Seller at channel sh is given as:

$$(s_1 s_2)\ s_1 \downarrow \mathsf{QuoteReq}(\mathtt{string}).\ s_2 \uparrow \mathsf{QuoteRep}(\mathtt{int}).\ s_1 \downarrow \mathsf{Purchase}(\mathtt{string}).\ \mathtt{0} \tag{23}$$

which says: firstly, two fresh session channels $s_1 s_2$ (in this order) are exchanged; then, using these two channels, communication of the represented shape takes place. Thus the service type (23) describes the whole of the behaviour starting from ch, albeit abstractly.

Example 9 (Session Type: branching) Let us refine (19) with branching.

$$\begin{array}{l} \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{QuoteReq}, prd, prd\rangle. \\ \mathsf{Seller} \to \mathsf{Buyer} : s_2\langle \mathsf{QuoteRep}, 100, y\rangle. \\ \left(\begin{array}{l} \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Purchase}, adr, adr\rangle.\mathbf{0} \\ \qquad + \\ \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Nothanks}\rangle.\mathbf{0} \end{array} \right) \end{array} \tag{24}$$

This can be abstracted, from the viewpoint of Seller:

$$\begin{array}{l} s_1 \downarrow \mathsf{QuoteReq}(\mathtt{string}).\ s_2 \uparrow \mathsf{QuoteRep}(\mathtt{int}). \\ \qquad (s_1 \downarrow \mathsf{Purchase}(\mathtt{string}).\mathtt{0}\ +\ s_1 \downarrow \mathsf{Nothanks}().\mathtt{0}) \end{array} \tag{25}$$

Note the sum + in (25) means the inputting party (here Seller) waits with two options, Purchase and Nothanks: on the other hand, the co-type of (25) (seen from Buyer's side) becomes:

$$\begin{array}{l} s_1 \uparrow \mathsf{QuoteReq}(\mathtt{string}).\ s_2 \downarrow \mathsf{QuoteRep}(\mathtt{int}). \\ \qquad (s_1 \uparrow \mathsf{Purchase}(\mathtt{string}).\mathtt{0}\ +\ s_1 \uparrow \mathsf{Nothanks}().\mathtt{0}) \end{array} \tag{26}$$

in which the sum + in (25) means that the outputting party (here Buyer) may select one of Purchase and Nothanks from the two options.

Example 10 (Session Type: recursion) Consider the following behaviour, in which

B continuously greets A.

$$\mathbf{rec}\ X.\mathsf{B} \to \mathsf{A} : s\langle \mathsf{Greeting}, \text{“}hello\text{”}, x\rangle.\ X \tag{27}$$

We can then abstract this behaviour as, from B's viewpoint:

$$\mathbf{rec}\ Y.\ s\ \uparrow\ \mathsf{Greetings}(\mathtt{string}).\ Y \tag{28}$$

whereas for A the same interaction is abstracted as:

$$\mathbf{rec}\ Y.\ s\ \downarrow\ \mathsf{Greetings}(\mathtt{string}).\ Y \tag{29}$$

which states that A repeatedly receives greetings. As a more meaningful recursion, consider the following refinement of (24):

$$\mathbf{rec}\ X.\left(\begin{array}{l}\mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{QuoteReq}, prd, prd\rangle.\\ \mathsf{Seller} \to \mathsf{Buyer} : s_2\langle \mathsf{QuoteRep}, 100, y\rangle.\\ \left(\begin{array}{l}\mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Purchase}, adr, adr\rangle.\mathbf{0}\\ \quad +\\ \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Nothanks}\rangle.\ X\end{array}\right)\end{array}\right) \tag{30}$$

This behaviour, seen from the viewpoint of Seller, can be abstracted as the following session type:

$$\mathbf{rec}\ Y.\left(\begin{array}{l}s_1 \downarrow \mathsf{QuoteReq}(\mathtt{string}).\\ s_2 \uparrow \mathsf{QuoteRep}(\mathtt{int}).\\ \left(\begin{array}{l}s_1 \downarrow \mathsf{Purchase}(\mathtt{string}).0\\ \quad +\\ s_1 \downarrow \mathsf{Nothanks}().Y\end{array}\right)\end{array}\right) \tag{31}$$

It may be notable that the following conditional has the same session type as (31).

$$\mathbf{rec}\ X.\left(\begin{array}{l}\mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{QuoteReq}, prd, prd\rangle.\\ \mathsf{Seller} \to \mathsf{Buyer} : s_2\langle \mathsf{QuoteRep}, 100, y\rangle.\\ \mathbf{if}\ \ \mathrm{reasonable}(y)@\mathsf{Buyer}\ \ \mathbf{then}\\ \quad \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Purchase}, adr, adr\rangle.\mathbf{0}\\ \mathbf{else}\\ \quad \mathsf{Buyer} \to \mathsf{Seller} : s_1\langle \mathsf{Nothanks}\rangle.\ X\end{array}\right) \tag{32}$$

One can further prefix (32) with a session initiation, for example with Buyer →

$\mathsf{Seller} : \mathsf{ch}(\mathsf{s}_1\mathsf{s}_2)$, in which case we obtain the service type for ch:

$$(s_1 s_2)\ \ \mathbf{rec}\ Y. \left(\begin{array}{l} s_1 \downarrow \mathsf{QuoteReq}(\mathtt{string}). \\ s_2 \uparrow \mathsf{QuoteRep}(\mathtt{int}). \\ \left(\begin{array}{l} s_1 \downarrow \mathsf{Purchase}(\mathtt{string}).0 \\ \qquad + \\ s_1 \downarrow \mathsf{Nothanks}().Y \end{array} \right) \end{array} \right) \tag{33}$$

which says that, after initialisation request exchanging two fresh session channels (designated as s_1 and s_2), it first waits for a $\mathsf{QuoteReq}$ message at s_1, to which it replies with $\mathsf{QuoteRep}$ via s_2, then it waits for two options $\mathsf{Purchase}$ and $\mathsf{Nothanks}$ at s_1: in the former case it finishes this session while in the latter it recurs to the initial state, waiting for another $\mathsf{QuoteReq}$ message.

4.3 Examples of Typing

Example 11 We conclude the section, by showing how it is possible to type an example: consider the buyer-seller case with the following interaction described in the global calculus.

$$\begin{array}{l} \mathtt{Buyer} \to \mathtt{Seller} : \mathtt{B2SCh}(s)\,.\,\mathtt{Buyer} \to \mathtt{Seller} : s\langle \mathtt{RequestForQuote} \rangle\,. \\ \quad \mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{QuoteResponse}, v_{\mathtt{quote}}, x_{\mathtt{quote}} \rangle\,. \\ \quad\ (\mathtt{Buyer} \to \mathtt{Seller} : s\langle \mathtt{QuoteReject} \rangle\ + \\ \quad\ \ \mathtt{Buyer} \to \mathtt{Seller} : s\langle \mathtt{QuoteAcceptance} \rangle\,. \\ \quad\ \ \ \mathtt{Seller} \to \mathtt{Shipper} : \mathtt{S2ShCh}(s')\,. \\ \quad\ \ \ \mathtt{Seller} \to \mathtt{Shipper} : s'\langle \mathtt{RequestDelDetails}, \mathtt{Buyer}, x_{\mathtt{Client}} \rangle\,. \\ \quad\ \ \ \ \mathtt{Shipper} \to \mathtt{Seller} : s'\langle \mathtt{DeliveryDetails}, \mathtt{DD}, x_{\mathtt{DD}} \rangle\,. \\ \quad\ \ \ \ \ \mathtt{Seller} \to \mathtt{Buyer} : s\langle \mathtt{DeliveryDetails}, x_{\mathtt{DD}}, x_{\mathtt{DD}} \rangle) \end{array}$$

Above there are two sessions: the one between the buyer and the seller, and the one between the seller and the shipper. Note that both are initialised by a session "init" operation and we have also included the choice. Another notable thing is that in the last two interactions, the variable $x_{\mathtt{DD}}$ is involved three times: the first two times it is indeed the same variable located at the seller and assigned with the delivery details $\mathtt{DD}$, but the third one is another variable located at the buyer which just happen to have the same name, but completely distinguished by the semantics of mini-CDL. But what are the types for channels $\mathtt{B2SCh}$ and $\mathtt{S2ShCh}$? It can be verified by the rules in [9] the interactions above can be typed by $\Delta = \mathtt{B2SCh@Seller}(s)[\mathtt{Buyer}, \mathtt{Seller}] : \alpha \cdot \mathtt{S2ShCh@Shipper}(s')[\mathtt{Seller}, \mathtt{Shipper}] : \alpha'$

where

$$\alpha = s \uparrow \texttt{RequestForQuote}().s \downarrow \texttt{QuoteResponse}(\texttt{QuoteType}).(s \uparrow \texttt{QuoteReject}() + s \uparrow \texttt{QuoteAcceptance}().s \downarrow \texttt{OrderConfirmation}().s \downarrow \texttt{DeliveryDetails}(\texttt{DDType}))$$

and $\alpha' = s' \uparrow \texttt{RequestDelDetails}(PType).s' \downarrow \texttt{DeliveryDetails}(\texttt{DDType})$.

5 A Larger Example and its Process Representation

This section demonstrates the expressiveness of the global calculus with a larger business protocol. The example is an extension of the Buyer-Seller protocol treated in Section 2 and comes from a use case discussed in [23]. After illustration of the protocol, we present the description of the protocol using the global calculus. We also show how this global description can be faithfully projected onto distributed communicating processes, written in a variant of the π-calculus, so that they realise precisely the original behaviour. The presentation of the projection procedure is informal, but it offers a glimpse of how a global description can be uniformly mapped to endpoint processes which faithfully realise the original behaviour.

5.1 Informal Description.

There are five participants involved in this protocol:

Buyer (B), Seller (S), Vendor (V), CreditChecker (CC) and RoyalMail (R).

The purpose of the protocol is for Buyer to ask for a quote of a product to Seller, negotiates the price, and buys the product if its price is cheap enough. Before asking for a quote, Buyer asks CreditChecker whether Seller is credible or not. The negotiation process is done as a loop, which involves not only Buyer and Seller but also Vendor (which only interacts with Seller). When the negotiation is successful, Seller asks CreditChecker if Seller is credible, and if the answer is positive, asks RoyalMail (a shipper) to ship the good. The following illustrates the protocol step by step.

(i) Buyer requests a service ch_{CC} for company check to the credit checker CreditChecker by sending its name.

(ii) At this point CreditChecker can either give a positive or negative answer.

(iii) If the answer is negative, the protocol terminates. If, on the other hand, the answer is positive, then the interactions enter the following negotiation loop:
  (a) Buyer asks Seller for a quote about product prod;
  (b) Seller then asks Vendor for service ch_V
  (c) Seller starts recursion and asks Vendor for a quote about product prod;
  (d) Vendor replies with a quote quote;
  (e) Seller forwards quote to Buyer increasing it by 10 units (quote+10);
  (f) if the quote is reasonable ($reasonable$(quote + 10)) then:

- Buyer sends Seller a confirmation (QuoteOK) together with the credit (cred);
- Seller then contacts CreditChecker for checking the credit;
- If the credit is good then:
  - Seller contacts RoyalMail (through the service channel ch_{Sh});
  - Seller sends the delivery address;
  - RoyalMail sends a confirmation;
  - Seller forwards confirmation to Buyer;
- If the credit is bad:
  - CreditChecker tells Buyer;
  - Buyer tells Seller terminating the protocol;

(g) if the quote is not reasonable the protocol goes back to point c;

This concludes the informal presentation of the protocol.

5.2 Representation in Global Calculus

We now give the formal description of this protocol using the global calculus. Since the description is long, we divide it into several parts. First we present a basic skeleton of the protocol.

1. $B \to CC : ch_{CC}(\boldsymbol{\nu} s)\,.\,CC \to B : s\langle \mathsf{Ack} \rangle\,.$
2. $B \to CC : s\langle CompanyCheck, sellerName, compName \rangle\,.$
3. $\{ \quad CC \to B : s\langle \mathsf{Good} \rangle\,.\,I_{\mathsf{Good}}$
4. $\quad +$
5. $\quad CC \to B : s\langle \mathsf{Bad} \rangle\,.\,\mathbf{0} \quad \}$

In Line 3, I_{Good} represents the interactions which take place after CreditChecker tells Buyer that the company is good, which is given as:

1. $B \to S : ch_S(\boldsymbol{\nu} t)\,.\,S \to B : t\langle \mathsf{Ack} \rangle\,.$
2. $B \to S : t\langle QuoteReq, prod, prod \rangle\,.$
3. $S \to V : ch_V(\boldsymbol{\nu} r)\,.\,V \to S : r\langle \mathsf{Ack} \rangle\,.$
4. **rec** $X\,.\,\{$
5. $\quad S \to V : r\langle QuoteReq, prod, prod \rangle\,.$
6. $\quad V \to S : r\langle QuoteRes, quote, quote \rangle\,.$
7. $\quad S \to B : t\langle QuoteRes, quote + 10, quote \rangle\,.$
8. $\quad$ if $reasonable(quote)@B$ then
9. $\quad\quad B \to S : t\langle QuoteOK, cred, cred \rangle\,.$
10. $\quad\quad S \to CC : ch_{CC}(\boldsymbol{\nu} u)\,.$
11. $\quad\quad CC \to S : u\langle \mathsf{Ack} \rangle\,.$
12. $\quad\quad S \to CC : u\langle PersonalCreditCheck, cred{:}adr, cred{:}adr \rangle\,.$
13. $\quad\quad \{ \quad CC \to S : u\langle \mathsf{Good} \rangle\,.\,I'_{\mathsf{Good}}$

14. $+$
15. $CC \to S : u\langle \mathsf{Bad} \rangle \,.$
16. $S \to B : t\langle \mathsf{YourCreditIsBad} \rangle \,.\, \mathbf{0} \quad \}$
17. else
18. $B \to S : t\langle \mathsf{QuoteNotOK} \rangle \,.\, X \quad \}$

Finally I'_{Good} in Line 13 above is given as:

1. $S \to R : ch_R(\boldsymbol{\nu} p) \,.$
2. $R \to S : p\langle \mathsf{Ack} \rangle \,.$
3. $S \to R : p\langle Deliv, adr, adr \rangle \,.$
4. $R \to S : p\langle \mathsf{Conf} \rangle \,.$
5. $S \to B : t\langle \mathsf{Conf} \rangle \,.\, \mathbf{0}$

This concludes the formal representation of the protocol.

5.3 Implementation of the Global Protocol

A global description such as above, when deployed, should be realised as communication processes located at each endpoint (participant). We now show how the global description above can be correctly implemented in each location. We shall informally use a distributed version of the π-calculus as we show in [9]. In particular, each participant will have a process P built on basic concurrency constructs.

- The construct $s \rhd \Sigma_{\mathsf{i}} \mathsf{op}_{\mathsf{i}}(x_i) \,.\, P$ denotes an input action on the session channel s with operations (options) op_i. The received value will be stored in one of the variables x_i according to which branch is selected.
- Dually, the construct $\overline{s} \lhd \mathsf{op}\langle e \rangle \,.\, P$ denotes the action of outputting the value e on the session channel s with operation op.
- The construct $!\, ch(\tilde{s}) \,.\, P$ is the input session initiation. It denotes the action of offering a replicated service ch.
- Dually, the construct $\overline{ch}(\nu \tilde{s}) \,.\, P$ is a request for the service ch (session initiation).
- The standard constructs $\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q$, $P \mid Q$, $P \oplus Q$ and $\mathbf{rec}\ X \,.\, P$ denote the the conditional operation, the parallel composition, a generic summation (not for input) and the recursion respectively.

For readability reasons, we will give different threads for each participants which will then be composed in parallel. Similarly to the global case, unimportant parameters in inputs and outputs will be omitted, e.g. $s \rhd \mathsf{op}$ stands for $s \rhd \mathsf{op}(x)$ for some x.

We start with Buyer's only thread:

$$\begin{array}{l}\overline{ch_{CC}}(\nu s)\,.\,s \rhd \mathsf{Ack}\,.\,\overline{s} \lhd \mathsf{CompanyCheck}\langle sellerName\rangle\,.\,s \rhd \\ \quad \{\ \mathsf{Good}\,.\,\overline{ch_S}(\nu t)\,.\,t \rhd \mathsf{Ack}\,.\,\overline{t} \lhd \mathsf{QuoteReq}\langle prod\rangle\,. \\ \quad\ \mathbf{rec}\ X\,.\ t \rhd \mathsf{QuoteRes}(quote)\,. \\ \quad\ \mathsf{if}\ reasonable(quote)\ \mathsf{then}\ \overline{t} \lhd \mathsf{QuoteOK}\langle cred\rangle\,. \\ \quad\ \ t \rhd \{\mathsf{YourCreditIsBad} + \mathsf{Conf}\} \\ \quad\ \mathsf{else}\ \overline{t} \lhd \mathsf{QuoteNotOK}\,.\,X \\ \quad\ + \\ \quad\ \mathsf{Bad}\quad \}\end{array}$$

Note this thread starts before the recursion and go through inside the (global) recursion. Thus this local behaviour also contains recursion. The next is a thread of CreditChecker.

$$\begin{array}{l}!\,ch_{CC}(s)\,.\,\overline{s} \lhd \mathsf{Ack}\,.\,s \rhd \mathsf{CompanyCheck}(compName)\,. \\ \quad \{\ \overline{s} \lhd \mathsf{Good}\ \oplus\ \overline{s} \lhd \mathsf{Bad}\ \ \}\end{array}$$

The symbol ! indicates replication i.e. the service ch_{CC} can be called an unbounded number of times. We now have another component of CreditChecker.

$$\begin{array}{l}!\,ch_{CC}(u)\,.\,\overline{u} \lhd \mathsf{Ack}\langle\rangle\,.\,u \rhd \mathsf{PersonalCreditCheck}(cred{:}adr)\,. \\ \quad \{\overline{u} \lhd \mathsf{Good}\ \oplus\ \overline{u} \lhd \mathsf{Bad}\ \ \}\end{array}$$

Note that both processes above do not include the recursion. This is because, in the global description, their sessions are inside a recursion so that they initiate a new instance of a session each time a recursion takes place. Furthermore, the two threads above describe the same service. Thus both interactions should come from the same process. Since, however, they differ in the second action, there should be a choice operation so that both possibilities are given. This results in:

$$\begin{array}{l}!\,ch_{CC}(u)\,.\,\overline{u} \lhd \mathsf{Ack}\,.\,u \rhd \\ \{\quad \mathsf{CompanyCheck}(cred{:}adr)\,.\,P_{\mathsf{gb}}\ +\ \mathsf{PersonalCreditCheck}(cred{:}adr)\,.\,P_{\mathsf{gb}}\quad \}\end{array}$$

where $P_{\mathsf{gb}} = \overline{u} \lhd \mathsf{Good} \oplus \overline{u} \lhd \mathsf{Bad}$. We now move to the end-point thread of Seller

$$
\begin{array}{l}
! \, ch_S(t) \,.\, \overline{t} \lhd \mathsf{Ack} \,.\, t \rhd \mathsf{QuoteReq}(prod) \,.\, \overline{ch_V}(\nu r) \,.\, t \rhd \mathsf{Ack} \,. \\
\quad \mathbf{rec}\ X \,.\ \overline{r} \lhd \mathsf{QuoteReq}\langle prod \rangle \,.\, r \rhd \mathsf{QuoteRes}(quote) \,. \\
\qquad \overline{t} \lhd \mathsf{QuoteRes}\langle quote + 10 \rangle \,.\, t \rhd \\
\qquad \{\ \mathsf{QuoteOK}(cred) \,.\, \overline{ch_{CC}}(\nu u) \,.\, u \rhd \mathsf{Ack} \,. \\
\qquad\quad \overline{u} \lhd \mathsf{PersonalCreditCheck}\langle cred{:}adr \rangle \,.\, u \rhd \\
\qquad\qquad \{ \mathsf{Good} \,.\, \overline{ch_R}(\nu p) \,.\, p \rhd \mathsf{Ack} \,. \\
\qquad\qquad\quad \overline{p} \lhd \mathsf{Deliv}\langle adr \rangle \,.\, p \rhd \mathsf{Conf} \,.\, \overline{t} \lhd \mathsf{Conf} \\
\qquad\qquad\quad + \\
\qquad\qquad\quad \mathsf{Bad} \,.\, \overline{t} \lhd \mathsf{CreditIsBad}\ \} \\
\qquad\quad + \\
\qquad\quad \mathsf{uoteNotOK} \,.\, X
\end{array}
$$

This thread starts outside of the recursion in the global description and carries through the loop, so that both the recursion and the recursion variable are used as they are, leading to the recursive behaviour of the process. The thread of Vendor follows.

$$
\begin{array}{l}
! \, ch_V(r) \,.\, \overline{t} \lhd \mathsf{Ack} \,. \\
\mathbf{rec}\ X \,.\, r \rhd \mathsf{QuoteReq}(prod) \,.\, \overline{r} \lhd \mathsf{QuoteRes}\langle quote \rangle \,.\, X
\end{array}
$$

Finally, we give the behaviour of RoyalMail.

$$
! \, ch_R(p) \,.\, \overline{p} \lhd \mathsf{Ack} \,.\, p \rhd \mathsf{Deliv}(adr) \,.\, \overline{p} \lhd \mathsf{Conf}
$$

A detailed formal analysis of a mapping of global descriptions to their endpoint representations are given in [9].

References

[1] Alur, R., K. Etessami and M. Yannakakis, *Realizability and verification of msc graphs*, Theoretical Computer Science **331** (2005), pp. 97–114.

[2] Baeten, J., H. van Beek and S. Mauw, *Specifying internet applications with DiCons*, in: *SAC'01* (2001), pp. 576–584.

[3] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C#*, ACM Trans. Program. Lang. Syst. **26** (2004), pp. 769–804.

[4] Berry, G. and G. Boudol, *The Chemical Abstract Machine*, TCS **96** (1992), pp. 217–248.

[5] Bhargavan, K., C. Fournet and A. Gordon, *Verified reference implementations of WS-Security protocols*, in: *WS-FN'06*, LNCS, 2006.

[6] Bonelli, E., A. B. Compagnoni and E. L. Gunter, *Correspondence assertions for process synchronization in concurrent communications.*, JFP **15** (2005), pp. 219–247.

[7] Briais, S. and U. Nestmann, *A formal semantics for protocol narrations*, in: *TGC*, 2005, pp. 163–181.

[8] Busi, N., R. Gorrieri, C. Guidi, R. Lucchi and G. Zavattaro, *Choreography and orchestration conformance for system design*, in: *Coodination*, LNCS **4038**, 2006, pp. 63–81.

[9] Carbone, M., K. Honda, N. Yoshida, R. Milner, G. Brown and S. Ross-Talbot, *A theoretical basis of communication-centred concurrent programming*, To be published by W3C. Available at http://www.dcs.qmul.ac.uk/$\sim$carbonem/cdlpaper/workingnote.pdf (2006).

[10] Dezani-Ciancaglini, M., D. Mostrous, N. Yoshida and S. Drossopoulou, *Session Types for Object-Oriented Languages*, in: *Proceedings of ECOOP'06*, LNCS, 2006.

[11] Gay, S. and M. Hole, *Subtyping for session types in the pi calculus*, Acta Informatica **42** (2005), pp. 191–225.

[12] Guidi, C., R. Lucchi, G. Zavattaro, N. Busi and R. Gorrieri, *Sock: a calculus for service oriented computing*, in: *ICSOC '06*, LNCS, 2006.

[13] Guttman, J. D., F. J. Thayer and L. D. Zuck, *The faithfulness of abstract protocol analysis: message authentication*, in: *CCS '01: Proceedings of the 8th ACM conference on Computer and Communications Security* (2001), pp. 186–195.

[14] Henriksen, J. G., M. Mukund, K. N. Kumar, M. A. Sohoni and P. S. Thiagarajan, *A theory of regular msc languages*, Information and Computation **202** (2005), pp. 1–38.

[15] Honda, K., V. Vasconcelos and M. Kubo, *Language primitives and type disciplines for structured communication-based programming*, in: *ESOP'98*, LNCS **1381**, 1998, pp. 22–138.

[16] Laneve, C. and L. Padovani, *Smooth orchestrators*, in: *FoSSaCS'06*, LNCS **3921**, 2006, pp. 32–46.

[17] Milner, R., *Functions as processes*, MSCS **2** (1992), pp. 119–141.

[18] Pierce, B. C., "Types and Programming Languages," MIT Press, 2002.

[19] Pierce, B. C. and D. N. Turner, *Pict: A programming language based on the pi-calculus*, in: *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, 2000 .

[20] Takeuchi, K., K. Honda and M. Kubo, *An interaction-based language and its typing system*, in: *PARLE'94*, LNCS **817**, 1994, pp. 398–413.

[21] van der Aalst, W., *Inheritance of interorganizational workflows: How to agree to disagree without loosing control?*, Info. Tech. and Management J. **2** (2002), pp. 195–231.

[22] Vasconcelos, V., A. Ravara and S. J. Gay, *Session types for functional multithreading.*, in: *CONCUR'04*, LNCS **3170**, 2004, pp. 497–511.

[23] W3C WS-CDL Working Group, *Web services choreography description language version 1.0*, http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/ .