A Theoretical Basis of Communication-Centred Concurrent Programming

Marco Carbone[1]   Kohei Honda[1]   Nobuko Yoshida[2]

[1]Queen Mary, University of London, UK
[2] Imperial College, London, UK

Abstract. We introduce two different ways of describing communication-centric software in the form of formal calculi and discuss their relationship. Two different paradigms of description, one centring on global message flows and another centring on local (end-point) behaviours, share the common feature, *structured representation of communications*. The global calculus originates from Choreography Description Language (CDL), a web service description language developed by W3C's WS-CDL Working Group. The local calculus is based on the π-calculus, one of the representative calculi for communicating processes. We illustrate these two descriptive frameworks using simple but non-trivial examples, present the static and dynamic semantics of these calculi, and show that any well-formed description (in a certain technical sense) in the global calculus has a precise representation in the local calculus.

1   Introduction

This paper introduces two different ways of describing communication-centred software in the form of formal calculi and discusses their relationship. Two different frameworks of description, one centring on global message flows and another centring on local (end-point) behaviours, share the common feature, *structured representation of communications*. The global calculus originates from Choreography Description Language (CDL) [2], a web service description language developed by W3C's WS-CDL Working Group. The local calculus is based on the π-calculus [4], one of the representative calculi for communicating processes. We show any well-formed description (in a technical sense we shall make clear) in the global calculus has a precise representation in the local calculus.

Both calculi are based on a common 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). As we shall explore in the present paper, the notion of session can be cleanly integrated with such notions as branching, recursion (loop) and exceptions. We show, through examples taken from simple but non-trivial business protocols, how concise structured description of non-trivial interactive behaviour is possible using sessions. From a practical viewpoint, a session gives us the following merits.

- It offers a clean way to describe a complex sequence of communications with rigorous operational semantics, allowing structured description of interactive behaviour.
- Session-based programs can use a simple, algorithmically efficient typing algorithm to check its conformance to expected interaction structures.
- Sessions offer a high-level abstraction for communication behaviour upon which further refined reasoning techniques, including type/transition/logic-based ones, can be built.

The presentation in this paper focusses the first point, and gives a formal basis for the second point. A full discussion of the second point and exploration of the third point are left to a later version of this paper and in its sequels.

An engineering background of the present work is the explosive growth of the Internet and world-wide web which has given rise to, in the shape of de facto standards, an omnipresent naming scheme (URI/URL), an omnipresent communication protocols (HTTP/TCP/IP) and an omnipresent data format (XML). These three elements arguably offer the key infra-structural bases for application-level distributed programming. 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 π-calculus and its theories of types are singular in that not only do they enable a study of diverse ways for structuring communication but also they allow fruitful and often surprising connections to existing formalisms including process algebras (e.g. CSP and CCS), functional computation (e.g. λ-calculus), logics (Linear Logic) and objects (e.g. Java). We believe a combination of strong practical needs for interactional computation and rich theoretical foundations will lead to rich dialogues between practice and theories. The present work is intended to offer some of the technical elements which may become useful in this dialogue.

This paper consists of two parts. In the first part, which are the first five sections (including this Introduction), we informally introduce two paradigms of describing interactions through incrementally complex examples. These examples come from use-cases for CDL found in CDL primer [5] by Steve Ross-Talbot and Tony Fletcher, and those communicated by Gary Brown [1] and Nickolas Kavantzas [3]. In the second part, which form the remaining sections, we introduce formal semantics, type discipline, and the formal connection between the core parts of these two formalisms.

Structure of the paper. In the rest of this paper, Sections 2, 3 and 4 are devoted to informal illustration of key technical elements through description of small but non-trivial use-cases in the global and local calculi. The description starts from a simple example and reaches a fairly complex one, illustrating the essence of each construct as well as the relationship between their respective global descriptions and the corresponding local ones. Section 5 comments on the correspondence and differences between our formal calculi and CDL. The second part (from Section 6 to Section 10) formally introduces two calculi (operational semantics in Section 6 and type disciplines in Section 7), develops theories of end-point projections (in Sections 8 and 9), and concludes the paper with related works and further topics (in Section 10). The appendix offers further technical details.
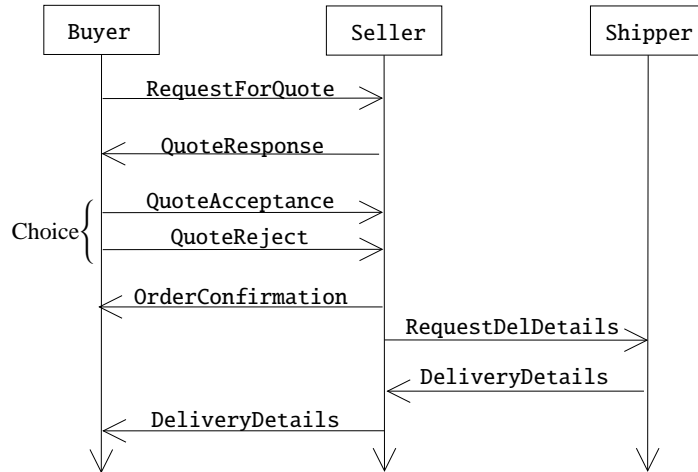
2 Describing Communication Behaviour (1)

Fig. 1. Graphical Representation of Simple Protocol

2.1 A Simple Business Protocol

In this section and the next, we show how small, but increasingly complex, business protocols can be accurately and concisely described in two small programming languages, one based on global message flows and another based on local, or end-point, behaviours. Along the way we also illustrate each construct of these mini programming languages (whose formal semantics is discussed in the second part of the paper).

Our starting point is a simple business protocol for purchasing a good among a buyer, a seller and a shipper, which we call **Simple BSH Protocol**. Informally the expected interaction is described as follows.

1. First, Buyer asks Seller, through a specified channel, to offer a quote (we assume the good to buy is fixed).
2. Then Seller replies with a quote.
3. Buyer then answers with either QuoteAcceptance or QuoteRejection. If the answer is QuoteAcceptance, then Seller sends a confirmation to Buyer, and sends a channel of Buyer to Shipper. Then Shipper sends the delivery details to Buyer, and the protocol terminates. If the answer is QuoteRejection, then the interaction terminates.

Figure 1 presents a UML sequence diagram of this protocol. Observe that, in Figure 1, many details are left unspecified: in real interaction, we need to specify, for example, the types of messages and the information exchanged in interaction, etc. While the protocol does not include practically important elements such as conditional and loops, its simplicity serves as a good starting point for introducing two formalisms.

2.2 Assumption on Underlying Communication Mechanisms

We first outline the basic assumptions common to both global and local formalisms. Below and henceforth we call the dramatis personae of a protocol (Buyer, Seller and Shipper in the present case), *participants*.

- We assume each participant either communicates through channels or change the content of variables local to it (two participants may have their own local variables with the same name but they are considered distinct).

- In communication:

  1. A sender participant sends a message and a receiver receives it, i.e. we only consider a point-to-point communication. A communication is always done through a *channel*. The message in a communication consists of an operator name and, when there is a value passing, a value. The value will be assigned to a local variable at the receiver's side upon the arrival of that message.

  2. Communication can be either an *in-session communication* which belongs to a session, or *session initiation communication* which establishes a session (which may be liked to establishing one or more fresh transport connections for a piece of conversation between two distributed peers). In a session initiation communication, one or more fresh session channels belonging to a session are declared, i.e. one session can use multiple channels.

  3. A channel can be either a *session channel* which belongs to a specific session or an *session-initiating channel* which is used for session-initiation. For a session-initiating channel, we assume its receiver is fixed.

- We may or we may not demand:

  1. the order of messages from one participant to another through a specified channel is preserved.

  2. each communication is synchronous, i.e. a sender immediately knows the arrival of a message at a receiver.

  3. one party participating in a session can use a session-channel both for sending and receiving.

The last three assumptions which we leave undermined do affect a way to formalise protocols, as well as for understanding their formal properties. Nevertheless the existence or lack of these assumptions do not substantially affect the informal discussions in this and the next section.

2.3 Representing Communication (1): Initiating Session

Buyer's session-initiating communication in Simple BSH Protocol is described in the global calculus as follows.

$$\mathsf{Buyer} \to \mathsf{Seller} : \mathsf{InitB2S}(\textit{B2Sch}).I \tag{1}$$

which says:

> Buyer initiates a session with Seller by communication through a session-initiating channel INITB2S, declaring a fresh in-session channel $\textit{B2S ch}$. Then interaction moves to I.

Note "**.**" indicates sequencing, as in process calculi. A session initiation can specify more than one session channels as needed, as the following example shows.

$$\mathsf{Buyer} \to \mathsf{Seller} : \mathsf{InitB2S}(\textit{B2Sch, S2Bch}).I \tag{2}$$

which declares two (fresh) session channels, one from Buyer to Seller and another in the reverse direction.

In local description, the behaviour is split into two, one for Buyer and another for Seller, using the familiar notation from process algebras. For example (1) becomes:

$$\mathsf{Buyer}[\ \mathsf{InitB2S}\,(\textit{B2Sch}).P_1\ ], \qquad \mathsf{Seller}[\ \overline{\mathsf{InitB2S}}\,(\textit{B2Sch}).P_2\ ] \tag{3}$$

Above Buyer[P] specifies a buyer's behaviour, while Seller[P] specifies a seller's behaviour. The overlined channel $\overline{\mathsf{InitB2S}}$ indicates it is used for output (this follows CCS/π-calculus: in CSP, the same action is written InitB2S!(B2Sch).

Note the behaviour of each participant is described rather than their interaction. When these processes are combined, they engage in interaction as described in (2).

2.4 Representing Communication (2): In-session Communication

An in-session communication specifies an operator and, as needed, a message content. First we present interaction without communication of values.

$$\mathsf{Buyer} \to \mathsf{Seller} : \textit{B2Sch}\,\langle\mathtt{QuoteRequest}\rangle.I' \tag{4}$$

where $\textit{B2S ch}$ is an in-session channel. It says:

> Buyer sends a *QuoteRequest*-message to Seller, then the interaction I' ensues.

The same behaviour can be written down in the local calculus as:

$$\overline{\textit{B2Sch}}\,\langle\mathtt{QuoteRequest}\rangle.P_1, \qquad \textit{B2Sch}\,\langle\mathtt{QuoteRequest}\rangle.P_2 \tag{5}$$

An in-session communication may involve value passing, as follows.

$$\mathsf{Seller} \to \mathsf{Buyer} : \textit{S2Bch}\,\langle\mathtt{QuoteResponse}, 3{,}000, x\rangle.I' \tag{6}$$

which says:

5

Seller sends a QuoteRespons-*message with value* 3,000 *to Buyer; Buyer, upon reception, assigns the received value,* 3,000, *to its local variable x.*

This description can be translated into end-point behaviours as follows.

$$\overline{S2Bch}\langle\text{QuoteResponse}, 3,000\rangle.P_1, \qquad S2Bch\langle\text{QuoteResponse}, y\rangle.P_2 \quad (7)$$

which describes precisely the same communication behaviour.

2.5 Representing Branching

In various high-level protocols, we often find the situation where a sender invokes one of the options offered by a receiver. A method invocation in object-oriented languages is a simplest such example. In a global calculus, we may write an in-session communication which involves such a branching behaviour as follows.

$$\{\text{Buyer} \rightarrow \text{Seller} : B2Sch\langle\text{QuoteAccept}\rangle.I_1\}$$
$$+ \qquad\qquad (8)$$
$$\{\text{Buyer} \rightarrow \text{Seller} : B2Sch\langle\text{QuoteReject}\rangle.I_2\}$$

which reads:

> Through an in-session channel B2Sch, Buyer selects one of the two options offered by Seller, QuoteAccept and QuoteReject, and respectively proceeds to I_1 and I_2.

The same interaction can be written down in the local calculus as follows. First, Buyer's side (the one who selects) becomes:

$$\{\overline{B2Sch}\langle\text{QuoteAccept}\rangle..P_1\}$$
$$\oplus \qquad\qquad (9)$$
$$\{\overline{B2Sch}\langle\text{QuoteReject}\rangle.P_2\}$$

Above $\oplus$ indicates this agent may either behave as $\overline{B2Sch}\langle\text{QuoteAccept}\rangle..P_1$ or $\overline{B2Sch}\langle\text{QuoteReject}\rangle.P_2$, based on its own decision (this is so-called *internal sum*, whose nondeterminism comes from its internal behaviour).

In turn, Seller's side (which waits with two options) becomes:

$$\{B2Sch\langle\text{QuoteAccept}\rangle.Q_1\}$$
$$+ \qquad\qquad (10)$$
$$\{B2Sch\langle\text{QuoteReject}\rangle.Q_2\}$$

Here + indicates this agent may either behave as $B2Sch\langle\text{QuoteAccept}\rangle.Q_1$ or as $B2Sch\langle\text{QuoteReject}\rangle.Q_2$ depending on what the interacting party communicates through B2Sch (this is so-called *external sum*, whose nondeterminism comes from the behaviour of an external process). Note both branches start from input through the same channel B2Sch.

In the local descriptions, the original sum in the global description in (8) is decomposed into the internal choice and the external choice. Similarly, I_1 (resp. I_2) may be considered as the result of interactions between P_1 and Q_1 (resp. P_2 and Q_2).

Buyer → Seller : InitB2S($B2Sch$).

Buyer → Seller : $B2Sch \langle$QuoteRequest$\rangle$.

Seller → Buyer : $B2Sch \langle$QuoteResponse, $v_{\mathrm{quote}}, x_{\mathrm{quote}}\rangle$.

{ { Buyer → Seller : $B2Sch \langle$QuoteAccept$\rangle$.

   Seller → Buyer : $B2Sch \langle$OrderConfirmation$\rangle$.

   Seller → Shipper : InitS2H($S2Hch$).

   Seller → Shipper : $S2Hch \langle$RequestDeliveryDetails$\rangle$.

   Shipper → Seller : $S2Hch \langle$DeliveryDetails, $v_{\mathrm{details}}, x_{\mathrm{details}}\rangle$.

   Seller → Buyer : $B2Sch \langle$DeliverDetails, $x_{\mathrm{details}}, y_{\mathrm{details}}\rangle$.**0** }

        +

{ Buyer → Seller : $B2Sch \langle$QuoteReject$\rangle$.**0** } }

Fig. 2. Global Description of Simple Protocol

2.6 Global Description of Simple BSH Protocol

We can now present the whole of a global description of Simple BSH Protocol, in Figure 2. While its meaning should be clear from our foregoing illustration, we illustrate the key aspects of the description in the following.

- Buyer initiates a session by invoking Seller through the session-initiating channel INITB2S, declaring an in-session channel $B2SCh$. Next, Buyer sends another message to Seller with the operation name "QuoteRequest" and without carried values (this message may as well be combined with the first one in practice).

- Seller then sends (and Buyer receives) a reply "QuoteResponse" together with the quote value v_{quote}. v_{quote} is a variable local to Seller (its exact content is irrelevant here). This received value will then be stored in x_{quote}, local to Buyer.

- In the next step, Buyer decides whether the quote is acceptable or not. Accordingly:
  1. Buyer may send QuoteAccept-message to Seller. Then Seller confirms the purchase, and asks Shipper for details of a delivery; Shipper answers with the requested details (say a delivery date), which Buyer forwards to Seller. Upon reception of this message the protocol terminates (denoted by **0**, the inaction).
  2. Alternatively Buyer may send QuoteReject-message to Seller, in which case the protocol terminates without any further interactions.

Remark. The description could have used more than one channels: for example, the Buyer-Seller interactions can use $S2Bch$ for communication from Seller to Buyer. The use of only $B2Sch$ may be considered as a way to describe "request-reply" mini-protocol inside a session, where an initial sender sends a request through a channel and a receiver in turn replies, in which case mentioning the initial channel suffices (which is a practice found in CDL, cf. [5]).

2.7 Local Description of Simple BSH Protocol

Figure 2 describes Simple BSH Protocol from a vantage viewpoint, having all participants and their interaction flows in one view. The same behaviour can be described focussing on behaviours of individual participants, as follows.

$$
\begin{array}{l}
\text{Buyer[} \overline{\text{InitB2S}}\,(B2Sch). \\
\qquad \overline{B2Sch}\,\langle\texttt{QuoteRequest}\rangle. \\
\qquad B2Sch\,\langle\texttt{QuoteResponse},\, x_{\text{quote}}\rangle. \\
\qquad \{\ \overline{B2Sch}\,\langle\texttt{QuoteAccept}\rangle. \\
\qquad\quad B2Sch\,\langle\texttt{OrderConfirmation}\rangle. \\
\qquad\quad B2Sch\,\langle\texttt{DeliveryDetails},\, y_{\text{details}}\rangle.\mathbf{0}\ \} \\
\qquad\qquad\qquad \oplus \\
\qquad \{\ \overline{B2Sch}\,\langle\texttt{QuoteReject}\rangle.\mathbf{0}\ \}\quad]
\end{array}
$$

$$
\begin{array}{l}
\text{Seller[InitB2S}\,(B2Sch). \\
\qquad B2Sch\,\langle\texttt{QuoteRequest}\rangle. \\
\qquad \overline{B2Sch}\,\langle\texttt{QuoteResponse},\, v_{\text{quote}}\rangle. \\
\qquad \{\ B2Sch\,\langle\texttt{QuoteAccept}\rangle. \\
\qquad\quad \overline{B2Sch}\,\langle\texttt{OrderConfirmation}\rangle. \\
\qquad\quad \overline{\text{InitS2H}}\,(S2Hch). \\
\qquad\qquad \overline{S2Hch}\,\langle\texttt{DeliveryDetails}\rangle. \\
\qquad\qquad S2Hch\,\langle\texttt{DeliveryDetails},\, x_{\text{details}}\rangle. \\
\qquad\quad \overline{B2Sch}\,\langle\texttt{DeliveryDetails},\, x_{\text{details}}\rangle.\mathbf{0}\ \} \\
\qquad\qquad\qquad + \\
\qquad \{\ B2Sch\,\langle\texttt{QuoteReject}\rangle.\mathbf{0}\ \}\quad]
\end{array}
$$

$$
\begin{array}{l}
\text{Shipper[InitS2H}\,(S2Hch). \\
\qquad S2Hch\,\langle\texttt{DeliveryDetails}\rangle. \\
\qquad \overline{S2Hch}\,\langle\texttt{DeliveryDetails},\, v_{\text{details}}\rangle.\mathbf{0}\quad]
\end{array}
$$

Fig. 3. Local Description of Simple Protocol

The description is now divided into (1) Buyer's interactive behaviour, (2) Seller's interactive behaviour, and (3) Shipper's interactive behaviour. One can intuitively see two descriptions of the same protocol, a global version in Figure 2 and a local version in Figure 3, represent the same software behaviours — we can extract the former from the latter and vice versa. We shall later establish such compatibility as a formal result. However there is a basic difference in the nature of descriptions: A global description allows us to see how messages are exchanged between participants and how, as a whole, the interaction scenario proceeds; whereas, in the local description, the behaviour of each party is made explicit, as seen in distinct forms of choices used in Buyer and Seller.

3 Describing Communication Behaviour (2)

3.1 Conditional

In Simple BSH Protocol, we only specified that Buyer may choose either `QuoteAccept` or `QuoteReject` nondeterministically. Suppose we wish to refine the description so that Buyer would choose the former when the quote is bigger than a certain amount, the latter if else. For this purpose we can use a conditional.

---

if $x_{\text{quote}} \leq 1000$ @Buyer **then**

{ Buyer $\rightarrow$ Seller : $B2Sch\langle$QuoteAccept$\rangle$.

   Seller $\rightarrow$ Buyer : $B2Sch\langle$OrderConfirmation$\rangle$.

   Seller $\rightarrow$ Shipper : InitS2H($S2Hch$).

   Seller $\rightarrow$ Shipper : $S2Hch\langle$RequestDeliveryDetails$\rangle$.

   Shipper $\rightarrow$ Seller : $S2Hch\langle$DeliveryDetails, $v_{\text{details}}, x_{\text{details}}\rangle$.

   Seller $\rightarrow$ Buyer : $B2Sch\langle$DeliverDetails, $x_{\text{details}}, y_{\text{details}}\rangle$.**0** }

 else

{ Buyer $\rightarrow$ Seller : $B2Sch\langle$QuoteReject$\rangle$.**0** }

Fig. 4. Global Description of Simple Protocol with Conditional

---

The 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. The same scenario is described as follows using the end-point calculus. Other participants' behaviours remain the same.

Buyer[$\overline{\text{InitB2S}}$ ($B2Sch$).

   $\overline{B2Sch}\langle$QuoteRequest$\rangle$.

   $B2Sch\langle$QuoteResponse, $x_{\text{quote}}\rangle$.

   if $x_{\text{quote}} \leq 1000$ **then**

   { $\overline{B2Sch}\langle$QuoteAccept$\rangle$.

      $B2Sch\langle$OrderConfirmation$\rangle$.

      $B2Sch\langle$DeliveryDetails, $y_{\text{details}}\rangle$.**0** }

    else

   { $\overline{B2Sch}\langle$QuoteReject$\rangle$.**0** }   ]

Fig. 5. Local Description of Simple Protocol with Conditional (Buyer)

3.2 Recursion.

Assume we wish to further refine the protocol with the following specification:

If the quote is too high, Buyer asks another quote until it receives a satisfactory quote.

Such behaviour is easily described using a loop or, more generally, recursion. In Figure 6, we show the global description of this enhanced protocol. There are only two

---

Buyer → Seller : InitB2S(*B2Sch*).

rec X.

{ Buyer → Seller : *B2Sch*⟨QuoteRequest⟩.

  Seller → Buyer : *B2Sch*⟨QuoteResponse, v_{quote}, x_{quote}⟩.

  if $x_{\text{quote}} \leq 1000$ @Buyer **then**

  { Buyer → Seller : *B2Sch*⟨QuoteAccept⟩.

    Seller → Buyer : *B2Sch*⟨OrderConfirmation⟩.

    Seller → Shipper : InitS2H(*S2Hch*).

    Seller → Shipper : *S2Hch*⟨RequestDeliveryDetails⟩.

    Shipper → Seller : *S2Hch*⟨DeliveryDetails, v_{details}, x_{details}⟩.

    Seller → Buyer : *B2Sch*⟨DeliverDetails, x_{details}, y_{details}⟩.**0** }

  else

  { Buyer → Seller : *B2Sch*⟨QuoteReject⟩.**X** } }

Fig. 6. Global Description of Simple Protocol with Conditional and Recursion

---

additional lines: in the second line, **rec X**. indicates that, intuitively:

We name the following block **X**. If **X** occurs inside that block, then we again recur to the top of the block.

In the last line, which is the second branch, **X** recurs again. Thus, at this point, the description recurs to a point immediately after **rec X** (i.e. the third line). The significance of recursion is its expressiveness (it can easily express various forms of loops) and its theoretical tractability. In the description, it is assumed that the variable v_{quote} will be updated appropriately by Seller, which is omitted from the protocol description.

    It is instructive to see how this recursion is translated into end-point behaviour. We present the local counterpart of Figure 6 in Figure 7 (we omit Shipper's behaviour which does not change). Observe both Buyer and Seller use recursion, so that they can collaboratively be engaged in recursive interactions. No change is needed in Shipper's local description, since it does not involve any recursion.

Buyer[$\overline{\mathsf{InitB2S}}\,(B2Sch)$.
       rec X.
       { $\overline{B2Sch}\,\langle\texttt{QuoteRequest}\rangle$.
         $B2Sch\,\langle\texttt{QuoteResponse},\,x_{\mathsf{quote}}\rangle$.
         if $x_{\mathsf{quote}} \leq 1000$ **then**
         { $\overline{B2Sch}\,\langle\texttt{QuoteAccept}\rangle$.
           $B2Sch\,\langle\texttt{OrderConfirmation}\rangle$.
           $B2Sch\,\langle\texttt{DeliveryDetails},\,y_{\mathsf{details}}\rangle.\mathbf{0}$ }
          else
         { $\overline{B2Sch}\,\langle\texttt{QuoteReject}\rangle.\mathbf{X}$ } }]

Seller[$\mathsf{InitB2S}\,(B2Sch)$.
       rec X.
       { $B2Sch\,\langle\texttt{QuoteRequest}\rangle$.
        $\overline{B2Sch}\,\langle\texttt{QuoteResponse},\,v_{\mathsf{quote}}\rangle$.
        { $B2Sch\,\langle\texttt{QuoteAccept}\rangle$.
         $\overline{B2Sch}\,\langle\texttt{OrderConfirmation}\rangle$.
         $\overline{\mathsf{InitS2H}}\,(S2Hch)$.
           $\overline{S2Hch}\,\langle\texttt{DeliveryDetails}\rangle$.
           $S2Hch\,\langle\texttt{DeliveryDetails},\,x_{\mathsf{details}}\rangle$.
         $\overline{B2Sch}\,\langle\texttt{DeliveryDetails},\,x_{\mathsf{details}}\rangle.\mathbf{0}$ }
               +
        { $B2Sch\,\langle\texttt{QuoteReject}\rangle.\mathbf{X}$ }]

Fig. 7. Local Description of Simple Protocol with Recursion (Buyer/Seller)

3.3 Timeout.

Let's consider refining Simple BSH protocol as follows:

> *If Buyer does not reply in 30 seconds after Seller presents a quote, then Seller will abort the transaction. Once Seller decides to do so, even if a confirmation message arrives from Buyer later, it is deemed invalid.*

For describing this refined behaviour, we first should have a means to describe a timeout. We consider this mechanism consisting of (1) creating a timer with a timeout value; (2) starting a timer; and (3) exception is thrown when a time out occurs. This exception is a *local exception*, in the sense that we consider our abstract notion of exceptions on the basis of the following infrastructural support:

> All exceptions in a participant are caught and handled within that participant in a specified local block (a participant may interact and compute normally as a result).

This is the standard, low-cost mechanism employed in many run-times such as those of Java and C++.

Let us see how this can be realised in concrete syntax. We first refine the global description in Figure 2. Some comments:

Buyer $\rightarrow$ Seller : InitB2S(*B2Sch, S2Babort*)**.**
    Buyer $\rightarrow$ Seller : *B2Sch* $\langle$QuoteRequest$\rangle$**.**
    Seller $\rightarrow$ Buyer : *B2Sch* $\langle$QuoteResponse, v_{quote}, $x_{\text{quote}}\rangle$**.**
    let t = **timer**(30)@Seller **in** {
        { Buyer $\rightarrow$ Seller : *B2Sch* $\langle$QuoteAccept$\rangle$ **timer**(t)**.**
        Seller $\rightarrow$ Buyer : *B2Sch* $\langle$OrderConfirmation$\rangle$**.**
        Seller $\rightarrow$ Buyer : *B2Sch* $\langle$DeliverDetails, x_{details}, $y_{\text{details}}\rangle$**.0** }
                +
        { Buyer $\rightarrow$ Seller : *B2Sch* $\langle$QuoteReject$\rangle$ **timer**(t)**.0** }
        catch (**timeout**(t))
        { Seller $\rightarrow$ Buyer : *S2Babort* $\langle$Abort$\rangle$**.0** } }

Fig. 8. Global Description of Simple Protocol with Timeout

- In the first line (initiating a session), two session channels, *B2Sch* (for default communications) and *S2Babort* (for aborting a transaction), are communicated through InitB2S. This generalised form of a session, where participants can use multiple channels in a single session, is useful for varied purposes.

- In the fourth line, a timer *t* with timeout value 30 is initiated at Seller. This timer will be stopped if the input guard specifying that timer (Lines 6 and 10) receives a message (the two branches of a single choice have the same timer).

- In the second line to the last, an exception handler is given, which says: when the timer fires, Seller will send an abort message to Buyer. It is omitted that, if Buyer's message arrives, Seller behaves as a sink, i.e. does nothing.

The same protocol can be described using the local formalism extended with timeout as follows. As before, in the exception branch, that Seller is assumed to behave as a

---

Seller[InitB2S *(B2Sch, S2Babort)*.
  B2Sch ⟨QuoteRequest⟩.
  $\overline{B2Sch}$ ⟨QuoteResponse, v_{quote}⟩.
  let *t* = **timer**(30) **in** {
    { *B2Sch* ⟨QuoteAccept⟩ **timer**(*t*).
      $\overline{B2Sch}$ ⟨OrderConfirmation⟩.
      $\overline{B2Sch}$ ⟨DeliveryDetails, x_{details}⟩.**0** }
        +
    { *B2Sch* ⟨QuoteReject⟩ **timer**(*t*).**0** }
    catch (**timeout**(*t*))
    { $\overline{S2Babort}$ ⟨Abort⟩abort.**0** } }]

Buyer[$\overline{\text{InitB2S}}$ *(B2Sch, S2Babort)*.
  {
    $\overline{B2Sch}$ ⟨QuoteRequest⟩.
    B2Sch ⟨QuoteResponse, x_{quote}⟩.
      { *B2Sch* ⟨QuoteAccept⟩.
        $\overline{B2Sch}$ ⟨OrderConfirmation⟩.
        $\overline{B2Sch}$ ⟨DeliveryDetails, x_{details}⟩.**0** }
          ⊕
      { $\overline{B2Sch}$ ⟨QuoteReject⟩.**0**
  }
  par
  { *S2Babort* ⟨Abort⟩abort.**0** }
]

Fig. 9. Local Description of Simple Protocol with Timeout

---

sink to messages at B2Sch (i.e. *B2Sch* ⟨QuoteAccept⟩.**0** + *B2Sch* ⟨QuoteReject⟩.**0** is omitted: it is possible it would behave non-trivially after it is in the abort mode). On the other hand, in Buyer's behaviour, we use **par** which indicates parallel composition. This behaviour is the same as before except the reception at the abort channel is added on parallel.

13

3.4 Combining Conditional, Recursion and Timeout

As a conclusion to this section, we present the combination of all constructs we have introduced so far. Figure Figure 10 gives a global description of the following behaviour:

Buyer → Seller : InitB2S(*B2Sch, S2Babort*).
rec X. {
   Buyer → Seller : *B2Sch* ⟨QuoteRequest⟩.
   Seller → Buyer : *B2Sch* ⟨QuoteResponse, v_{quote}, x_{quote}⟩.
   let t = **timer**(30)@Seller **in** {
     if (x_{quote} ≤ 1000 @Buyer) {
       Buyer → Seller : *B2Sch* ⟨QuoteAccept⟩ **timer**(t).
       Seller → Buyer : *B2Sch* ⟨OrderConfirmation⟩.
       Seller → Shipper : InitS2H(*S2Hch*).
       Seller → Shipper : *S2Hch* ⟨RequestDeliveryDetails⟩.
       Shipper → Seller : *S2Hch* ⟨DeliveryDetails, $v_{details}$, $x_{details}$⟩.
       Seller → Buyer : *B2Sch* ⟨DeliverDetails, $x_{details}$, $y_{details}$⟩.
       0
     } **else** {
       Buyer → Seller : *B2Sch* ⟨QuoteReject⟩ **timer**(t).**X** }
     catch (**timeout**(t)) {
       Seller → Buyer : *S2Babort* ⟨Abort⟩.**0**
     }
   }
}

Fig. 10. Global Description of BSH Protocol with Conditional/Loop/Timeout

1. First, Buyer asks Seller, through a specified channel, to offer a quote (we assume the good to buy is fixed).
2. Then Seller replies with a quote.
3. Buyer then answers with either "I will buy" (if the price is cheap) or "I will not buy" (if not) to Seller. S
4. If the answer is "I will buy", then Seller sends a confirmation to Buyer, and sends a channel of Buyer to Shipper. Then Shipper sends the delivery details to Buyer, and the protocol terminates.
5. If the answer is "I will not buy", then the interaction recurs to (1) above.
6. If Buyer does not reply in time, Seller will abort the transaction.

The local description is given in Figure 11.

14

Buyer[$\overline{\mathsf{InitB2S}}$ (*B2Sch, S2Babort*).
    {
       rec X.
       { $\overline{B2Sch}\,\langle\texttt{QuoteRequest}\rangle$.
         $B2Sch\,\langle\texttt{QuoteResponse}, x_{\mathsf{quote}}\rangle$.
         if $x_{\mathsf{quote}} \leq 1000$ **then**
         { $\overline{B2Sch}\,\langle\texttt{QuoteAccept}\rangle$.
          $B2Sch\,\langle\texttt{OrderConfirmation}\rangle$.
          $B2Sch\,\langle\texttt{DeliveryDetails}, y_{\mathsf{details}}\rangle.\mathbf{0}$ }
          else
         { $\overline{B2Sch}\,\langle\texttt{QuoteReject}\rangle.\mathbf{X}$ } }
       par
       $S2Babort\,\langle\texttt{ABORT}, x_{\mathsf{abort}}\rangle.\mathbf{0}$
    }
  ]

Seller[$\mathsf{InitB2S}$ (*B2Sch, S2Babort*).
    rec X.
    { $B2Sch\,\langle\texttt{QuoteRequest}\rangle$.
     $\overline{B2Sch}\,\langle\texttt{QuoteResponse}, v_{\mathsf{quote}}\rangle$.
     let $t = \mathbf{timer}(30)$ **in** {
     { $B2Sch\,\langle\texttt{QuoteAccept}\rangle\,\mathbf{timer}(t)$.
      $\overline{B2Sch}\,\langle\texttt{OrderConfirmation}\rangle$.
      $\overline{\mathsf{InitS2H}}$ (*S2Hch*).
       $\overline{S2Hch}\,\langle\texttt{DeliveryDetails}\rangle$.
       $S2Hch\,\langle\texttt{DeliveryDetails}, x_{\mathsf{details}}\rangle$.
      $\overline{B2Sch}\,\langle\texttt{DeliveryDetails}, x_{\mathsf{details}}\rangle.\mathbf{0}$ }
           +
     { $B2Sch\,\langle\texttt{QuoteReject}\rangle\,\mathbf{timer}.\mathbf{X}$ }
     catch (**timeout**(*t*))
     { $\overline{S2Babort}\,\langle\texttt{Abort}, abort\rangle.\mathbf{0}$ }
    }
  ]

Shipper[$\mathsf{InitS2H}$ (*S2Hch*).
     $S2Hch\,\langle\texttt{DeliveryDetails}\rangle$.
     $\overline{S2Hch}\,\langle\texttt{DeliveryDetails}, v_{\mathsf{details}}\rangle.\mathbf{0}$]

Fig. 11. End-Point Description of BSH Protocol with Conditional/Loop/Timeout

4 Describing Communication Behaviour (3)

4.1 Criss-Crossing of Actions: Proactive Quoting (1)

In this section we treat behaviours which involve *criss-crossing*: between two participants, say A and B, one message goes from A to B and another from B to A in parallel, one of which often having a stronger priority. We use use-cases contributed by Gary Brown [1] and Nickolas Kavantzas [3].

Brown's use-case is a (simplified form of) one of the typical interaction patterns in Investment Bank and other businesses. Its narrative description is extremely short, but the induced behaviour is non-trivial to describe. We assume two participants, A and B.

1. Initially, A sends a request for quote to B.
2. Then B sends an initial quote to A as a response.
3. Then B will enter a loop, sending pro-actively a new quote in a "RefreshQuote"-message every 5 seconds until A's "AcceptQuote"-message arrives at B.

Thus the "AcceptQuote"-message from A is in a race condition with a "RefreshQuote"-message from B. Once the quote is accepted, B should terminate its loop. We leave unspecified in the use-case how a quote is calculated, how A decides to accept a quote, and how A notifies which quote A is agreeing on (refinements are easy).

The repeated actions at each time interval can be cleanly modelled using the predicate-based invocation mechanism [2], which is also useful for other purposes.

$$\textbf{when } (p@A) \{I\}$$

where p is a predicate (an expression of a boolean type). It reads:

> The interaction I does not start until the predicate p becomes true: when it becomes so, then I will be engaged in.

Its precise semantics is either (1) whenever p becomes true, I should start; or (2) when p becomes true, I can start, but this "event" can be missed in which case I may not start. The behaviour in (1) tends to become more deterministic, while (2) is realisable through busy-waiting without additional synchronisation mechanism.

We use this construct to describe the use-case. We first informally illustrate the underlying idea (suggested by [1]): after the initial quote has arrived at A, we consider there are two independent threads of interactions, in both A and B.

– In one, A may decide to send the "AcceptQuote"-message; when B receives it, B will set its local variable $p_{\texttt{quoteAccepted}}$ to "truth" (which should be initially "false").
– In another, A is always ready to receive "RefreshQuote"-message (with a new quote value); On the other hand, *as far as the local variable $p_{\texttt{quoteAccepted}}$ is false*, B will repeatedly send, at each 5 seconds, a fresh quote.

Note the variable $p_{\texttt{quoteAccepted}}$ is used for communication between two threads in B. When B ceases to send new quotes, A also ceases to react to new quotes from B, thus both reaching a quiescent state. The description in the global formalism (augmented with "when"-construct) follows.

16

$\mathsf{A} \to \mathsf{B} : \mathsf{InitA2B}(\textit{A2Bch}, \textit{B2Ach}).$

$\mathsf{A} \to \mathsf{B} : \textit{A2Bch} \langle \mathtt{RequestQuote} \rangle.$

$\mathsf{B} \to \mathsf{A} : \textit{A2Bch} \langle \mathtt{Quote}, y_{\mathsf{quote}}, x_{\mathsf{quote}} \rangle.$

$p_{\mathsf{quoteAccepted}} = \mathtt{ff} @ B.$

{

  $\tau_A . \mathsf{A} \to \mathsf{B} : \textit{A2Bch} \langle \mathtt{AcceptQuote} \rangle. \ p_{\mathsf{quoteAceppted}} := \mathtt{tt} @ B. \ \mathbf{0}$

  par

  rec $X.$ {

    let $t = \mathbf{timer}(5)@B$ **in**

    when $(\mathtt{expired}(t)@B)$

      if$(p_{\mathsf{quoteAccepted}} = \mathtt{ff}@B)$ { $\mathsf{B} \to \mathsf{A} : \textit{B2Ach} \langle \mathtt{RefreshQuote}, y_{\mathsf{quote}}, x_{\mathsf{quote}} \rangle.\mathbf{X}$ }

}

Fig. 12. A Proactive Quoting with a Criss-Cross (global)

Above, "τ_A" is the standard τ-action local to A, indicating passage of an unspecified duration of time. Thus as a whole

$$\tau_A . \mathsf{A} \to \mathsf{B} : \textit{A2Bch} \langle \mathtt{AcceptQuote} \rangle. \ p_{\mathsf{quoteAceppted}} := \mathtt{tt} @ B. \ \mathbf{0},$$

indicates that the sending of "AcceptQuote" (with a quote value at the time) may take place after some duration of time, and when B receives this message, B will assign "truth" to its local variable $p_{\mathsf{quoteAceppted}}$. One may as well refine the above part as follows, using the "when" construct.

$$\mathbf{when} \ (\mathtt{satisfied})@A$$
$$\{ \mathsf{A} \to \mathsf{B} : \textit{A2Bch} \langle \mathtt{AcceptQuote} \rangle. \ p_{\mathsf{quoteAceppted}} := \mathtt{tt} @ B. \ \mathbf{0} \}$$

where `satisfied` is an unspecified predicate local to A, indicating the satisfaction of A w.r.t., say, the current quote value.

In the second thread, B is engaged in a loop: the timer t expires at each 5 seconds and, when $\mathtt{expired}(t)$ (which is a predicate rather than exception) becomes true, the body of "when" is executed. If $p_{\mathsf{quoteAceppted}}$ is false, it sends a quote and re-enters the loop: if $p_{\mathsf{quoteAceppted}}$ is true, it terminates the loop. The interaction

$$\mathsf{B} \to \mathsf{A} : \textit{B2Ach} \langle \mathtt{RefreshQuote}, y_{\mathsf{quote}}, x_{\mathsf{quote}} \rangle$$

not only indicates B sends a "RefreshQuote"-message, but also A is ready to receive it and sets the communicated quote into its variable x_{quote}.

The protocol description invites us to diverse forms of refinement. For example, we may consider the predicate `satisfied` is a boolean variable set after A receives a new quote (in the second thread). We leave exploration of such refinements to the reader.

A[$\overline{\mathsf{InitA2B}}$ *(A2Bch, B2Ach)*.
   $\overline{A2Bch}\,\langle\texttt{RequestQuote}\rangle$.
   B2Ach $\langle\texttt{Quote}, x_{quote}\rangle$.
   $\{\,\tau.\overline{A2Bch}\,\langle\texttt{AcceptQuote}, x_{quote}\rangle.\mathbf{0}$ **par** **rec X**.$\{\,B2Ach\,\langle\texttt{RefreshQuote}, x_{quote}\rangle.\mathbf{X}\,\}\,\}$
]

B[InitA2B *(A2Bch, B2Ach)*.
   A2Bch $\langle\texttt{RequestQuote}\rangle$.
   $\overline{B2Ach}\,\langle\texttt{RefreshQuote}, y_{\texttt{quote}}\rangle$.
   $p_{\texttt{quoteAcceppted}} := \texttt{ff}$.
   $\{$
      A2Bch $\langle\texttt{AcceptQuote}\rangle.\ p_{\texttt{quoteAcceppted}} := \texttt{tt}$. $\mathbf{0}$
      par
      rec *X*. $\{$
      let $t = $ **timer**(5) **in when** $(\texttt{expired}(t))$
         $\{$ **if** $(p_{\texttt{quoteAccepted}} = \texttt{ff})\ \{\ \overline{B2Ach}\,\langle\texttt{RefreshQuote}, y_{quote}\rangle.\mathbf{X}\ \}\ \}$
   $\}$
]

Fig. 13. A Proactive Quoting with a Criss-Cross (local)

Next we consider the local version of Figure 12, using the end-point counterpart of the "when"-construct. This is given in Figure 13. One may compare the presented behaviours with those in Figure 12. The "when" construct is used in *B*, with the same semantics as in the global calculus.

In the local description of *A*'s behaviour, the projection makes clear that, in one of its two threads, *A* repeatedly gets ready to receive "RefreshQuote"-messages from *B*, while, independently, may move to the stage where it sends an "AcceptQuote"-message to *B*. Thus, when a criss-cross of these messages take place, *A* will simply receives the message from *B* while sending its own message. As noted before, we may as well refine *A*'s behaviour, for example in its transition to the quote acceptance state.

In the local description of *B*, the first thread does not start from the τ-action (which is *A*'s local action) but starts from the reception of "QuoteAcceptance"-message from *A*. The second thread is engaged with the timeout and loop using the "when" construct, using the variable $p_{\texttt{quoteAccepted}}$.

The local descriptions of the proactive quoting protocol in Figure 13 are directly related with its global description in Figure 12 and vice versa, up to the treatment of criss-crossing. In particular, it is not hard to imagine how we can project the description in Figure 12 to the one in Figure 13 following a simple principle. A natural question is whether we can do the *reverse* translation in a general way: can we integrate the local descriptions in Figure 13 to synthesize the global description in Figure 12? What would be the general principle involved in these two kinds of translations? Part of this question will be answered in Part II of the present paper.

4.2 Criss-Crossing of Actions: Proactive Quoting (2)

In this subsection, we present an alternative global description of the proactive quoting protocol. It is simple and understandable, even though the description is only sound under a strong assumption about the underlying communication mechanism.

$A \rightarrow B$: InitA2B(*A2Bch, B2Ach*).

$A \rightarrow B$: *A2Bch* ⟨RequestQuote⟩.

$B \rightarrow A$: *B2Ach* ⟨Quote, quote, x_{Quote}⟩.

rec *X*.{

   let t = **timer**(5)@*B* **in**

     $A \rightarrow B$: *A2Bch* ⟨AcceptQuote⟩**timer**(t)**.0**

   catch(*timeout*(t))

     $B \rightarrow A$: *B2Ach* ⟨RefreshQuote, *newQuote*, x_{Quote}⟩**.X** }

Fig. 14. A Proactive Quoting with a Criss-Cross (global, with atomic interaction)

The description in Figure 14 is terse and understandable. Its soundness however depends on a significant condition on the underlying messaging semantics: *each interaction is atomic*. This assumption becomes essential in $A \rightarrow B$: *A2Bch* ⟨AcceptQuote⟩ (the sixth line). If this interaction were not executed atomically, it would be possible that *A* sends a AcceptQuote-message to *B*, but the time-out in *B* is caught, *B* sends RefreshQuote to *A*, and *A* again sends AcceptQuote-message, but the usecase says *A* does so only once. Under the assumption of atomicity, however, the interaction $A \rightarrow B$: *A2Bch* ⟨AcceptQuote⟩ either happens or not at all and moves to a timer, which is only realisable if this action is atomic. It may be costly to realise such atomicity in general. At the same time, the description may suggest atomicity of interaction can lead to terse specification of a complex behaviour.

Due to the assumption on atomicity and its interplay with timer, it is hard to devise local descriptions directly corresponding to Figure 14. Even if we stipulate the same atomicity assumption in local descriptions, it is hard to construct the projection onto *A*: the problem is that the 'when' loop within *A* does not have an activity that it can observe to indicate that *B* has exited the loop. A possible approach to this would be to model a guard condition for *A* to also include the 'quoteAccepted' variable — but this guard condition would also have to include the aspect of duration, otherwise (as a result of the 'when' blocking semantics) the guard at *A* would simply block until the variable 'quoteAccepted' was set to true, and it would not receive any of the quote refresh messages. Further, if both participants are required to use the same guard condition, then it also assumes they have synchronised clocks and evaluate the expressions at exactly the same time.

4.3 Criss-Crossing of Actions: A T-Shirts Procurement Protocol (1)

Next we treat Kavantzas's use-case [3], which describes a protocol for purchase orders between a really big corporation (RBC) and a small T-shirts company (STC).

1. RBC sends a purchase order (PO) to STC.
2. STC acknowledges the PO and initiates a business process to handle the PO.
3. After STC's internal processes regarding the PO are completed, STC sends "PO-Completed" to RBC in order for RBC to complete its own business process.
4. RBC can send a Cancel Order message to abort STC's business process (which can criss-cross with a PO completed message), any time before RBC receives the PO Completed message from STC
5. If Cancel Order arrives at STC before PO Completed is sent from STC, then STC aborts its business process and acknowledges this to RBC with PO Cancelled, in order for RBC to abort its own business process. Otherwise, if STC has already sent PO Completed, it ignores the Cancel Order because RBC has agreed it will honor POs when cancellations are not sent out within an agreed-upon time-frame.
6. If RBC has already sent the Cancel Order message and then it receives the PO Completed message, then instead of aborting, RBC completes it.

Figure 15 presents a global description of this protocol.

$RBC \rightarrow STC : \mathsf{InitR2S}(R2Sch)\mathtt{.}$

$RBC \rightarrow STC : R2Sch\langle\mathtt{CreateOrder}\rangle\mathtt{.}$

$STC \rightarrow RBC : R2Sch\langle\mathtt{OrderAck}\rangle\mathtt{.}$

$\mathbf{let}\ t = \mathbf{timer}(T)@RBC\ \mathbf{in}\ \{$

   $\{STC \rightarrow RBC : R2Sch\langle\mathtt{POCompleted}\rangle\ \mathbf{timer}(t)\mathtt{.0}\}$

$\mathbf{catch\ timeout}(t)\ \{$

  $RBC \rightarrow STC : S2Rabort\langle\mathtt{Abort}\rangle\mathtt{.}\{$

    $STC \rightarrow RBC : R2Sabort\langle\mathtt{ConfirmAbort}\rangle\mathtt{.0}$

    $+$

    $STC \rightarrow RBC : R2Sch\langle\mathtt{POConfirmation}\rangle\mathtt{.0}\ \ \}\}\}$

Fig. 15. A Global Description of T-Shirts Procurement

Above, RBC first initialises a session channel R2Sch through InitR2S, then sends an order, which STC acknowledges. RBC then starts a timer, i.e. the longest time T it is willing to wait before the PO confirmation arrives. The timer is frozen upon the PO confirmation. Alternatively if the time-out occurs, it is handled by the catch part: RBC sends an abort message to STC, and either STC acknowledges it or its PO-confirmation arrives. Note we have made a timer explicit in this description: we later show a description which does not rely on the use of a timer.

An acute reader may observe that this description again assumes atomicity of communication, as in the previous subsection, in the sense that the execution of an interaction A → B : $ch\langle\mathtt{Op}\rangle$ indicates the two things at the same time: *A* sends a message and *B* has received that message.

Next we give an end-point counterpart of the same description, in Figure 16.

RBC[$\overline{\mathsf{InitR2S}}$ (*R2Sch, S2Rch*).
      $\overline{R2Sch}\langle\mathtt{CreateOrder}\rangle$.
      $S2Rch\langle\mathtt{OrderAck}\rangle$.
      let $t = \mathbf{timer}(T)$ **in**   {
         $S2Rch\langle\mathtt{POCompleted}\rangle\mathbf{timer}(t).\mathbf{0}$
      }
      catch timeout(t)   {
         $\overline{S2Babort}\langle\mathtt{Abort},\mathbf{true}\rangle$.
            $S2Babort\langle\mathtt{ConfirmAbort}\rangle.\mathbf{0}$
            +
            $S2Rch\langle\mathtt{POCompleted}\rangle.\mathbf{0}$}   ]

STC[$\mathsf{InitR2S}$ (*R2Sch, S2Rch*).
      $R2Sch\langle\mathtt{CreateOrder}\rangle$.
      $x_{\mathtt{Abort}} := \mathbf{false}$.
      $\overline{S2Rch}\langle\mathtt{OrderAck}\rangle$.
      try
      { $\tau.\overline{S2Rch}\langle\mathtt{POCompleted}\rangle.\mathbf{0}$}
      catch ($\neg x_{\mathtt{Abort}}$)
         {$\overline{S2Babort}\langle\mathtt{ConfirmAbort}\rangle.\mathbf{0}$
         +
         $\overline{S2Rch}\langle\mathtt{POCompleted}\rangle.\mathbf{0}$}
      par
      $S2Babort\langle\mathtt{Abort}, x_{\mathrm{abort}}\rangle.\mathbf{0}$]

Fig. 16. A Local Description of T-Shirts Procurement

In STC's description, we use the following predicate-based exception mechanism. The syntax for this exception handling is:

$$\mathbf{try}\ \{P\}\ \mathbf{catch}\ (p)\ \{Q\}$$

whose semantics is, informally: to execute the interaction *P* unless the predicate (a boolean-valued expression) *p* is satisfied (note *p* is treated as an event). In the latter case, *Q* would be executed. This construct is feasibly implemented if the "catch" part is an exception such as timeout or explicitly thrown exceptions. However its implementation becomes more involved if, as here, a predicate is used for invocation since in that case a mechanism is necessary to watch the update of relevant variables. Note this construct

is similar to the "when" construct: the same underlying mechanism can realise both. As an alternative, one may realise a similar behaviour using either a busy-waiting or a "sleep" construct, though these alternatives may not be faithful to intended semantics when we use arbitrary predicates for invocation.

We illustrate the behaviour of RBC and STC in this end-point description. First, RBC's local behaviour is as follows.

– The first three actions (session initialisation, order request and acknowledgement) are just as before;
– RBC sets a timer and waits for T time-units to receive the PO confirmation from STC;
– If the time-out is triggered, RBC will send an abort to STC, and then wait for the abort confirmation or for the PO confirmation.

The local behaviour of STC may be illustrated thus.

– As in the RBC part, the first three actions need no description, apart the fact that STC has a variable for checking whether RBC has requested an abort or not. This variable is initialised to false;
– At this point STC checks the abort variable, and if it is not true it decides to perform a tau action and then send the PO confirmation.
– if the abort variable is true it then confirms the abort;
– in parallel with the described thread, there is another thread which just waits for an abort message from RBC.

Note the end-point description makes it explicit how timeout is done and how criss-crossing occurs in terms of two distributed end-point behaviours. We believe it faithfully realises the global behaviour described in Figure 15 under the assumption of atomicity of interactions: at the same time, one may observe that the given end-point description might not be automatically extracted from the global description. In fact, as far as the initial protocol description goes, the local description arguably realises a correct behaviour even if we do not stipulate the atomicity of communication (it is notable that interactions in CDL [2] as a default does not assume atomicity, though its "align" mechanism is closely related with atomic execution of communication).

4.4 Criss-Crossing of Actions: A T-Shirts Procurement Protocol (2)

The descriptions so far depend on the explicit use of timer and exception (timeout) which a timer engenders. However the nondeterminism and criss-crossing of message exchanges themselves may not be directly related with local use of timers. Indeed, a description of the overall exchange of interactions is possible without using timers, as we shall discuss below.

The protocol uses two (local) variables, AbortRequested at STC and ConfArrived at RBC, both initialised to be false. The timing of update of these variables is the key underlying idea of this protocol. Let us offer an informal illustration of the protocol.

– The initial three interactions remain the same as before, i.e. sending a purchase order from RBC to STC after a session initiation, then an acknowledgement from STC to RBC.

RBC → STC : InitR2S(*R2Sch, S2Rabort*).
RBC → STC : *R2Sch*⟨CreateOrder⟩.
STC → RBC : *R2Sch*⟨OrderAck⟩.
{

   $x_{\texttt{AbortRequested}}@STC :=$ **false.**

   τ_{STC} **.**

   if $\neg x_{\texttt{AbortRequested}}@STC$ {

     STC → RBC : *R2Sch*⟨POConfirmation⟩.

     $x_{\texttt{ConfArrived}}@RBC :=$ **true.0**}

   else

     STC → RBC : *R2Sabort*⟨ConfirmAbort⟩.**0**

}

par

{

   $x_{\texttt{ConfArrived}}@RBC :=$ **false.**

   τ_{RBC} **.**

   if $\neg x_{\texttt{ConfArrived}}@RBC$ {

     RBC → STC : *S2Rabort*⟨Abort⟩.

     $x_{\texttt{AbortRequested}} :=$ **true.0** }

}

Fig. 17. A Global Description of T-Shirts Procurement without Timer

– At this stage the interactions are divided into the parallel composition of two behaviours. In one thread of interaction, we have:

1. STC will, at some point, check AbortRequested is true (i.e. RBC's abort request has arrived) or false (i.e. RBC's abort request has not arrived).
2. If AbortRequested is false, then STC will send a PO confirmation message. When RBC receives it, it will set its ConfArrived to be true, and STC moves to the completion of PO processing.
3. If AbortRequested is true, then STC will send a AbortConfirmed message. RBC receives it, and in both sites the PO process aborts.

In another thread of interaction, we have:

1. At some point RBC will check ConfArrived.
2. If it is false (i.e. a PO confirmation has not arrived), then sends AbortRequest-message to STC.
3. If it is true (i.e. a PO confirmation has arrived), then RBC moves to the completion of PO processing.

In Figure 17, τ_{STC} (resp. τ_{RBC}) indicates a τ-action in STC (resp. in RBC), which may take an unspecified amount of time. We can check that this protocol never moves to:

– The situation where STC sends a PO confirmation but an RBC aborts (since, for an RBC to abort, it needs to obtain AbortConfirm message from STC).
– The situation where RBC receives both a PO-confirmation and AbortConfirm (since the output of these two messages by STC are mutually exclusive).

Note however it is possible STC may receive, in one thread, AbortRequest message at time t but, for some reason, this has not been propagated to another thread in time, so that, at time $t + t_0$, STC sends a PO-confirmation message to RBC. However this does not contradicts the initial specification (we also believe this is consistent with the standard business convention).

The end-point projection of this example is not to hard, which we leave to the reader. We also note Kavantzas [3] presents a different description in CDL using the "when" construct with distributed predicates.

4.5   Further Note

In this section we have explored various ways to describe two business protocols (though the presented ones are far from the only ways to describe them). The purpose of these formal representations of business protocols in formal calculi is not only to analyse the behaviours of these protocols themselves and to reason about them, but also to understand the correspondence between various constructs and their expressiveness. By having a precise operational semantics, we can discuss diverse aspects of the constructs needed to represent a large class of communication behaviours with precision. Further analyses of these and other complex business protocols in these formalisms would be an important and stimulating future research topic.

5 Correspondence with CDL

In this section, we briefly outline relationship between CDL and the global/local calculi we have used in the previous sections. The correspondence/differences are summarised in Table 1.

feature	CDL	formalism
session channels	located at input	no restriction
session initiation	implicit	explicit
general co-relation	yes	by adding "polyadic sync"
typing	by-name (informal)	by-structure (formal)
type checking	no	yes
local exception	none	yes
repetition	loop	recursion
sequencing	imperative	prefix
predicate-based invocation	yes	by adding "when"
EPP	implemented	proved
global variable lookup	yes	no
global completion	yes	no

Table 1. Correspondence and Differences

Some comments:

- Channels are one of the fundamental elements in communication-based languages as well as in security engineering, arising in diverse forms (such as sockets, remote object IDs, and URLs). Even though an informal global description may not mention channels (this is because the names of participants play the role of channels), they become essential when exception and channel passing are involved. In fact, in standard distributed programming, we may use multiple channels (often in the shape of transport connections) in one unit of conversation.
- CDL channels are located at the inputting side, representing the ports where the sender writes to. Formalisms are more general, using channels both for input and for output. (as we noted before, CDL allows both-way usage of a channel when we specify a pair of request and reply with one channel).
- Concerning session initiation, this is done implicitly in CDL. In our calculi, we place the explicit session initiation which makes the underlying operational and type structure more explicit and more amenable to analyses. This does not prevent us from using the calculus to represent practical business protocols since we may regard the session initiation and the subsequent action to be combined into a single message in implementation.
- Co-relation is one of the significant features of CDL. Co-relation can be considered as a way to collectively treat multiple sessions as one conversation unit. Though we have not treated in this work, this feature can be cleanly represented in formal calculi. One method is to use the so-called polyadic synchronisation.

- CDL does not have a proper notion of type checking nor type inference. However it is equipped with such notions as relationship, roles and participants, whose specifications are related with each other through XML schemas. These constructs play an important role as part of documentation. These data will be usable as a basis of typing, using the so-called by-name approach (as found in Java).

- In the current CDL specification, type checking (i.e. verifying if a particular choreography is well typed) is not part of the specification. Such type checking may as well be partly complemented by type inference (i.e. elaborating untyped phrases with appropriate types). These verifications can be done formally in the calculus, i.e. we can provide an algorithm which, given an interaction I and a type t, checks whether the t is a good type for I. Transporting this facility into a CDL development tool will be one of the significant future topics.

- As we saw above, exception are indispensable for describing many real-world protocols. One thing missing in WS-CDL would be the ability to handle exceptions locally, with a standard local scoping rule. This topic may deserve further consideration (CDL has an exception specific to a sub-choreography, which may serve part of this functionality).

- Repetition of instructions is usually dealt with while loops. In the calculus we use recursion, another mechanism which can faithfully emulate the standard loop operation as well as many forms of recursive calls. They also enjoy many theoretical features. This does not mean it is better to replace loops with recursion: when a loop behaviour is intended, writing it with a loop often leads to a more understandable program.

- Sequencing of interactions can be treated in two different ways, i.e. the way it is done in CDL and the way it is done in π-calculus. In CDL, a standard imperative language construct ";" is adopted. In our formalisms, we are using the simple prefixing operator. Superficially, the latter construct is less powerful than the former, mainly because it assumes only very simple operations are allowed before the ".". On the contrary, when using ";" we can combine complex expressions such as those combined by the parallel operator. Again there is a precise embedding of ";" into the prefixing in combination with other constructs, so we lose no generality in using "." while allowing easier analysis.

- CDL is equipped with the predicate-driven invocation mechanism (for which we used the construct **when**). This mechanism is powerful for various specifications, but it also demands a heavy implementation mechanism. Exploration of cases where this construct becomes indispensable would become important for understanding its status in structured concurrent programming.

- Various globalised features of CDL are incorporated because they often naturally arise in business protocols. Their semantic content however may not be precisely understood. Note globalised behaviour has to be, in effect, realised by interactions among distributed peers. Therefore, at least at the level of formalisms, the understanding of how a certain global construct may be realised by interactions is a prerequisite for their proper inclusion in formalisms. Precise appreciation of what high-level global abstraction would be suitable for describing communication-centred software behaviour, and how they relate to their local (communication-based) realisation, is an important topic for future study.

References

1. G. Brown. A post at pi4soa forum. October, 2005.
2. W3C Web Services Choreography Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.
3. N. Kavantzas. A post at petri-pi mailing list. August, 2005.
4. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
5. S. Ross-Talbot and T. Fletcher. Ws-cdl primer. Unpublished draft, November, 2005.