Session-based Choreography with Exceptions

Marco Carbone[1,2]

Department of Computer Science
Queen Mary University of London
London, United Kingdom

Abstract

Choreography has recently emerged as a pragmatic and concise way of describing communication-based systems such as web services and financial protocols. Recent studies have investigated the transition from the design stage of a system to its implementation providing an automatic way of mapping a choreograhy into executable code.
In this work, we focus on an extension of choreography with a communication-based (interactional) exception mechanism by giving its formal semantics. In particular, we discuss through some examples how interactional exceptions at choreography level can be implemented into end-point code.

1 Introduction

Due to fast-evolving technologies, applications based on communication are becoming vital in the practical world. Such communication-centred applications are often called *Web Services*, the first major programming trend which positions communication as a key element of high-level programming.

An emerging paradigm for programming communication is the so-called *Choreography*. This discipline focuses on the fact that an architect, when designing a distributed system, no longer describes the behaviour of each single interacting peer (*end-point behaviour*) but establishes how the various inter-participant interactions happen by giving a *global description* (choreography) of the system. In a traditional approach, the architect would describe each communication operation, e.g. an input to be performed at one particular peer. Unfortunately, this makes it very difficult to have a global view of how the whole system being designed works. On the other hand, a global description offers a view of where and when a communication has to happen. The architect will now decide that e.g. there will be a message from A to B and no longer think how this will be implemented at A (sending a message) or B (waiting to receive a message). Choreography can offer a vantage view of the

[1] I would like to thank Kohei Honda, Nobuko Yoshida and the anonymous referees for their helpful suggestions. This work is partially supported by EP/F002114.

[2] Email: carbonem@dcs.qmul.ac.uk

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

system facilitating the design stage and leaving the implementation details to the (possibly automated) process of generating a sound end-point code, called *end-point projection*.

Exceptions are a mechanism widely adopted in modern programming languages (e.g. Java, C#) for dealing with exceptional system behaviours i.e. they are designed to handle the occurrence of some conditions interrupting the normal flow of execution of a program [3]. While the classical notion of exception is bound to the local flow of a process, in communication-centred programming exceptions are about the flow of interactions where a sudden interruption must involve all interacting participants. We shall call this kind of exception an *interactional exception* [5].

This work studies an extension of choreography with exceptions in the setting of session-based communication and session types [6]. In particular, we shall see how exceptions are naturally interactional in choreography (choreography is about interactions) while they require an exception propagation mechanism at end-point level.

Contributions of the paper. Below, we list the contributions of this paper:

- syntax, reduction semantics and typing of the global calculus with exceptions;
- syntax, reduction semantics and typing of the end-point calculus with exceptions and locations;
- a discussion through examples of the requirements for guaranteeing a sound mapping from the global calculus to the end-point calculus (end-point projection).

Outline. The remainder of the paper is divided as follows: Section 2 introduces the extension of the global calculus with exception primitives and discusses the new typing system where a new type for exceptions, the try-catch type is introduced; Section 3 adapts the results found in [5] to a calculus with locations; Section 4 discusses with examples the end-point projection and how it changes in presence of exception constructs; Section 5 concludes the paper.

2 Extending Choreography with Exceptions

Syntax. The global calculus [3,4] is a model of choreography based on WS-CDL [4] [10]. We hereby extend it with new terms for describing exceptions. The syntax of a *global description* (or choreography) I is defined as:

$$
\begin{array}{rlrlrl}
I, J ::= & A \to B : a(s)[\tilde{t}, I, J] & \text{(init)} & \mid & \mathbf{0} & \text{(inaction)} \\
\mid & A \to B : s\langle \mathsf{op}, e, y\rangle \,.\, I & \text{(com)} & \mid & I \mid J & \text{(par)} \\
\mid & \mathbf{throw} & \text{(throw)} & \mid & I + J & \text{(sum)} \\
\mid & \mathbf{if}\ e@A\ \mathbf{then}\ I\ \mathbf{else}\ J & \text{(cond)} & \mid & X & \text{(recVar)} \\
\mid & \mathbf{rec}\ X \,.\, I & \text{(rec)} & & &
\end{array}
$$

[3] Note that exceptions is not necessarily associated to unwanted/erroneous behaviour: *"Exception (handling) is a programming language construct or computer hardware mechanism designed to handle the occurrence of a condition that changes the normal flow of execution"* - Wikipedia

[4] WS-CDL is a description language developed by W3C WS-CDL Working Group based on the notion of choreography and it is mainly used for business protocols.

$a, b, c, \ldots$ range over service channels which may be considered as public services; s, t, r, u range over session channels, the communication channels freshly generated for each session; A, B, C range over participants who are equipped with their local variables denoted by $x, y, z \ldots$; op is a label denoting an operation (as in standard session types [6]); e is an arithmetic or other first order expression containing variables; and X ranges over term variables used for recursion. The free session channels and term variables of a global description I (denoted by $\mathsf{fn}(I)$ and $\mathsf{fv}(I)$ respectively) are defined in the usual way.

Most of the terms are similar to [3]. (com) models in-session communication between participants A and B: the result of evaluating message e at A will be stored in variable y located at B while op indicates the type of operation required in the interaction (e.g. a method) [10]. (cond) is the standard conditional operator where the guard e is evaluated at A. (rec) and (recVar) are respectively recursion and recursion variable while (sum) is non-deterministic choice. The term (par) is the parallel composition of choreographies.

In the syntax above, the terms (throw) and (init) are novel. (throw) denotes the throwing of an exception. (init) describes a system where participant A invokes a service b located at participant B and starts a session s (in the rest of the paper $A \neq B$). The novelty is in the triple $[\tilde{t}, I, J]$: choreography I, called the *default choreography*, describes the normal (or default) behaviour of the system, while J is the *exception handler* i.e. the way the system shall behave when an exception is thrown. Vector $\tilde{t}$, containing already established sessions, has a pivotal rôle: if there is a throw in I then any other handler belonging to an embedding triple $[\tilde{t}', I', J']$ ($[\tilde{t}, I, J]$ is in I') such that $\tilde{t}' \subseteq \tilde{t}$ must be discarded. We call vector $\tilde{t}$ a *refinement* in the sense that channels $\tilde{t}'$ in $\tilde{t}$ are refined i.e. a new handler J replaces the old ones. We also let s itself be included in $\tilde{t}$ which is convenient for typing. As an example, the choreographies

- $A \to B : b(s)[s, \quad B \to C : c(t)[t, I, J] \quad , J']$
- $A \to B : b(s)[s, \quad B \to C : c(t)[(s, t), I, J] \quad , J']$

describe different systems. In the first case, the raising of an exception in I, would only concern the interactions on t between B and C while the second choreography would also affect the interactions on s. In the latter case, a throw in I would execute handler J discarding J'. Note that, in the implementation, this condition means that an exception on any t_i has to be propagated to all the elements of $\tilde{t}$. Unfortunately, a refinement (s, t, r) inside an outer default choreography with refinement (s, t, t') would create an inconsistency as the latter says that an exception should be propagated over s, t and t' while the inner refinement would only consider propagation over s and t in addition to the new t'. For consistency, we make the following assumption:

Assumption 1 *For any* $C \to D : a(s')[\tilde{t}', I', J']$ *occurring in* $A \to B : b(s)[\tilde{t}, I, J]$ *and for any* $t_i \in \tilde{t}$ *we have* $t_i \in \tilde{t}'$ *implies* $\tilde{t} \subseteq \tilde{t}'$.

For having consistent operational semantics, we also stipulate the following syntactic constraints: (i) recursions should be guarded, i.e. I in $\mathbf{rec}\ X\,.\,I$ is prefixed by (init) or (com); and (ii) in $A \to B : b(s)[\tilde{t}, I, J]$, a free term variable never occurs

in I or J. Further, in $A \longrightarrow B : b(s)[\tilde{t}, I, J]$, we shall exclude refinements on t_i in J. As well, we do not allow top-level throws in handlers. This prevents a handler from throwing a further exception in the same session.

Example 2.1 *(A simple Financial Protocol)* This example [5] is a typical scenario in financial protocols. Consider a customer Buyer who wishes to purchase a product from a company Seller. Buyer starts a session with Seller who repeatedly sends quote updates about the product price. When Buyer decides to accept a particular quote, without explicitly notifying Seller, it throws an exception. At this point, Seller and Buyer move together to a new stage (exceptional stage with respect to the ordinary sequence of actions) where they exchange information for successfully terminating the transaction e.g. credit card details for payment and receipt. A global calculus representation of the protocol which uses the interactional exception mechanism is given:

$$
\begin{array}{ll}
1. & \texttt{Buyer} \rightarrow \texttt{Seller} \ : \ \textbf{chSeller}(s)\ [s, \\
2. & \quad \textbf{rec}\ X\,.\texttt{Seller} \rightarrow \texttt{Buyer} : s\langle \texttt{update}, \mathit{quote}, y\rangle\,. \\
3. & \quad \textbf{if}\ (y < 100)@\texttt{Buyer}\ \textbf{then throw else}\ X, \\
4. & \quad \texttt{Seller} \rightarrow \texttt{Buyer} : s\langle \texttt{conf}, \mathit{cnum}, x\rangle\,. \\
5. & \quad \texttt{Buyer} \rightarrow \texttt{Seller} : s\langle \texttt{data}, \mathit{credit}, x\rangle\ ]
\end{array}
$$

lines 2–3: default; lines 4–5: handler

In line 1, Buyer invokes service **chSeller** from Seller. Line 2 and 3 compose the default choreography: the interaction $\texttt{Seller} \rightarrow \texttt{Buyer} : s\langle \texttt{update}, \mathit{quote}, y\rangle$ models the sending of a quote *quote* from Seller to Buyer who will store the received value at variable y. In line 3, variable y is checked by Buyer and if its value is less than 100, an exception will be thrown otherwise the course of action will go back to line 2. Lines 4 and 5 describe how the system will handle an exception: Seller will send a confirmation *cnum* and Buyer will reply with its credit card *credit*. Note that variable x denotes two different variables in lines 4 and 5: the first one is located at Buyer while the second is located at Seller.

The behaviour described by the choreography above may also be represented by replacing the throw operator with the code in the handler, i.e.:

$$
\begin{array}{l}
\textbf{rec}\ X\,.\texttt{Seller} \rightarrow \texttt{Buyer} : s\langle \texttt{update}, \mathit{quote}, y\rangle\,. \\
\textbf{if}\ (y < 100)@\texttt{Buyer}\ \textbf{then}\ \{\ldots \text{as in handler} \ldots\}\ \textbf{else}\ X,
\end{array}
$$

Although this is a legitimate way of describing the protocol, the choreography above increases the burden of the implementation. An end-point representation would require a hack i.e. that Buyer notifies Seller of its guard evaluation at each iteration while our solution can be implemented with end-point interactional exceptions thus automatically guaranteeing the propagation of the exceptional state. Hence, the user (or architect) could be allowed to write exception-free descriptions as the above but then a tool would have to translate them into something like the one with exceptions.

Example 2.2 *(A Financial Protocol with Broker)* We extend the protocol above including a third participant, a broker Broker whose rôle is to buy from Seller and resell to Buyer (a typical situation in financial protocols). In this scenario, Buyer will invoke Broker rather than Seller and act almost identically as in the previous example. On the other hand Broker, after being invoked by Buyer and checking his reputation, will invoke Seller and act as Buyer in the previous example. As before, Buyer can raise an exception in case of quote acceptance but also Broker can throw if Buyer's reputation is not satisfactory before even invoking Seller. This can be written in the global calculus as:

1. $\texttt{Buyer} \to \texttt{Broker} : \mathbf{chBroker}(s)\ [s,$
2. $\quad \texttt{Buyer} \to \texttt{Broker} : s\langle \texttt{identify}, id, x\rangle\,.$
3. $\quad \mathbf{if}\ \texttt{bad}(x)@\texttt{Broker}\ \mathbf{then\ throw}$
4. $\quad \mathbf{else}\ \texttt{Broker} \to \texttt{Seller} : \mathbf{chSeller}(t)[(s,t),\ \mathbf{rec}\ X\,.$
5. $\quad\quad \texttt{Seller} \to \texttt{Broker} : t\langle \texttt{update}, quote, y\rangle\,.$
6. $\quad\quad \texttt{Broker} \to \texttt{Buyer} : s\langle \texttt{update}, 1.1 * y, y\rangle\,.$ } default
7. $\quad\quad \mathbf{if}\ (y < 100)@\texttt{Buyer}\ \mathbf{then\ throw\ else}\ X,$ }
8. $\quad\quad \texttt{Seller} \to \texttt{Broker} : t\langle \texttt{conf}, cnum, x\rangle\,.$
9. $\quad\quad \texttt{Broker} \to \texttt{Buyer} : s\langle \texttt{conf}, x, x\rangle\,.$
10. $\quad\quad \texttt{Buyer} \to \texttt{Broker} : s\langle \texttt{data}, credit, x\rangle\,.$ } handler
11. $\quad\quad \texttt{Broker} \to \texttt{Seller} : t\langle \texttt{data}, x, x\rangle],$ } default
12. $\quad \texttt{Broker} \to \texttt{Buyer} : s\langle \texttt{reject}, reason, x\rangle\ ]$ } handler

In lines 1 and 2, Buyer invokes service **chBroker** and sends its identity *id* to Broker who, in line 3, will check whether Buyer is bad or not. If Buyer is not trusted, Broker will raise an exception which will take both Buyer and Broker to an abortion procedure in line 12. Note that in this case, Buyer and Broker are the only participants involved so far and the only ones who will move to another conversation for handling the exception.

If Buyer can be trusted, Broker invokes service **chSeller** and forwards to Buyer all quotes received from Seller increasing them by 10%. As before, Buyer will throw an exception whenever s/he decides to accept a quote. In this case, as the participants involved are now Buyer, Broker and Seller, the handler to be executed is the inner one where Broker will forward messages between Buyer and Seller (see lines 8-11). This event will also discard the handler in line 12 which, after session initiation in line 4, has become inactive.

Semantics. Above, we have shown how interactional exceptions can be exploited

for designing systems such as financial protocols. Now, we formalise the exception mechanism by defining a reduction semantics for the global calculus that can handle exceptions.

In order to give semantics to the new exception operations, we need to extend the syntax with the following terms (called run-time syntax):

$$I, J \quad ::= \quad \ldots \quad | \quad \mathbf{try}(\tilde{s})\ \{\, I \,\}\ \mathbf{catch}\ \{\, J \,\} \quad | \quad \{\!\{I\}\!\} \quad | \quad (\boldsymbol{\nu} s)\ I$$

The term on the left, called *try-catch block*, reads: "The system is behaving as choreography I; if an exception is thrown the system will start behaving as described by choreography J". This realises at run-time the default choreography and the handler found in session initialisation. When an exception is thrown, we need to make sure that handlers do not get brutally terminated by an embedding try-catch block. This is ensured by a *wrap* term $\{\!\{I\}\!\}$ which reads "an exception has been thrown and part of the system is behaving as I". Term $(\boldsymbol{\nu} s)\ I$ is the standard restriction. In addition, as terms may reduce inside try-catch blocks, we need a notion of (run-time) context, defined as:

$$C[-] \ ::= \mathbf{try}(\tilde{s})\ \{\, C[-] \,\}\ \mathbf{catch}\ \{\, J \,\} \ \mid\ I \mid C[-] \ \mid\ \{\!\{C[-]\}\!\} \ \mid\ (\boldsymbol{\nu} s)\ C[-] \ \mid\ -$$

Table 1 contains the rules generating the reduction relation $\longrightarrow$. A reduction $(I, \sigma) \xrightarrow{\tilde{s}} (J, \sigma')$ says that a global description I in a state σ (which is the collection of all local states of the participants) evolves into description J with a new state σ'. Intuitively, the reduction semantics of choreography will change the state σ by updating local variables as a consequence of each interaction. Label $\tilde{s}$ is used for discarding embedding try-catch blocks (due to refinement) on some s_i after an exception has been thrown. We shall often omit the label when equal to the empty set $\emptyset$.

We now discuss the reported rules. Rule (G-Init) transforms session initialisation into a try-catch term i.e. the term $A \to B : a(s)[\tilde{t}, I, J]$ is rewritten into $(\boldsymbol{\nu} s)\ (\mathbf{try}(\tilde{t})\ \{\, I \,\}\ \mathbf{catch}\ \{\, J \,\})$ denoting the generation of the session channel s (now under restriction) and the use of a try-catch block. The state σ remains unchanged. Rule (G-Com) is about intra-session communication: participant A is sending message v (the result of evaluating expression e in the A-part of state σ) over session channel s and B will receive and assign v to its local variable x. As a result of this communication, state σ will become $\sigma[x@B \mapsto v]$. Rule (G-Thr) handles the raising of an exception: when a throw is top-level in a try-catch block then the default choreography terminates and the handler J is run (wrapped). The rule uses a different relation $I \searrow (I', \tilde{s})$ called meta reduction. The initial choreography I is transformed into I', the result of erasing and wrapping nested try-blocks and catch-blocks respectively. $\tilde{s}$ denotes all those session channels affected by the exception from nested try-catch blocks. This relation is indispensable in interactional exceptions: choreography I may contain wraps or other try-catch blocks which should not be brutally erased and an exception should also be raised. Meta reduction is the minimum relation $\searrow$ defined by the rules in Table 2. (G-MTry) is the most relevant rule. Whenever a try-catch block must be meta reduced, we need to apply meta reduction to its default choreography first. If this contains a refinement of $\tilde{s}$

$$\text{(G-Init)}\ \frac{-}{(\sigma,\ A \twoheadrightarrow B : a(s)[\tilde{t}, I, J]) \longrightarrow (\sigma, (\boldsymbol{\nu} s)\ \mathbf{try}(\tilde{t})\ \{\, I \,\}\ \mathbf{catch}\ \{\, J \,\})}$$

$$\text{(G-Com)}\ \frac{\sigma' = \sigma[x@B \mapsto v] \quad \sigma \vdash e@A \Downarrow v}{(\sigma,\ A \to B : s\langle \mathsf{op}, e, x\rangle \,.\, I) \longrightarrow (\sigma',\ I)}$$

$$\text{(G-Thr)}\ \frac{\mathbf{try}(\tilde{t})\ \{\, \mathbf{throw} \mid I \,\}\ \mathbf{catch}\ \{\, J \,\} \searrow (I', \tilde{s})}{(\sigma, \mathbf{try}(\tilde{t})\ \{\, \mathbf{throw} \mid I \,\}\ \mathbf{catch}\ \{\, J \,\}) \xrightarrow{\tilde{t} \cup \tilde{s}} (\sigma, I')}$$

$$\text{(G-Clean)}\ \frac{\tilde{s} \subseteq \tilde{t}, \quad (\sigma, I) \xrightarrow{\tilde{t}} (\sigma', I') \quad \text{and} \quad I' \searrow (I'', \tilde{r})}{(\sigma, \mathbf{try}(\tilde{s})\ \{\, I \,\}\ \mathbf{catch}\ \{\, J \,\}) \xrightarrow{\tilde{t} \cup \tilde{r}} (\sigma, I'')}$$

$$\text{(G-Context)}\ \frac{(\sigma, I) \xrightarrow{\tilde{s}} (\sigma', I') \qquad \tilde{s} \not\subseteq \mathsf{fn}(C[-])}{(\sigma, C[I]) \xrightarrow{\tilde{s}} (\sigma', C[I'])} \qquad \text{(G-Res)}\ \frac{(\sigma, I) \xrightarrow{\tilde{s}} (\sigma, J)}{(\sigma, (\boldsymbol{\nu} t)\ I) \xrightarrow{\tilde{s} \setminus \{t\}} (\sigma, (\boldsymbol{\nu} t)\ J)}$$

$$\text{(G-IfT)}\ \frac{\sigma \vdash e@A \Downarrow \mathsf{tt}}{(\sigma,\ \mathbf{if}\ e@A\ \mathbf{then}\ I_1\ \mathbf{else}\ I_2) \longrightarrow (\sigma, I_1)} \qquad \text{(G-Sum)}\ \frac{}{(\sigma,\ I_1 + I_2) \longrightarrow (\sigma,\ I_1)}$$

$$\text{(G-IfF)}\ \frac{\sigma \vdash e@A \Downarrow \mathsf{ff}}{(\sigma,\ \mathbf{if}\ e@A\ \mathbf{then}\ I_1\ \mathbf{else}\ I_2) \longrightarrow (\sigma, I_1)} \qquad \text{(G-Par)}\ \frac{(\sigma,\ I_1) \longrightarrow (\sigma',\ I_1')}{(\sigma, I_1 \mid I_2) \longrightarrow (\sigma',\ I_1' \mid I_2)}$$

$$\text{(G-Rec)}\ \frac{(\sigma,\ I\{\mathbf{rec}\ X.I/X\}) \longrightarrow (\sigma',\ I')}{(\sigma,\ \mathbf{rec}\ X.I) \longrightarrow (\sigma',\ I')} \qquad \text{(G-Struct)}\ \frac{I \equiv I'' \ \ (\sigma,\ I) \longrightarrow (\sigma',\ I') \ \ I' \equiv I'''}{(\sigma,\ I'') \longrightarrow (\sigma',\ I''')}$$

Table 1
Semantics Rules for the Global Calculus with Exceptions

$$\text{(G-MTry)}\quad I \searrow (I', \tilde{t}) \quad \Rightarrow \quad \mathbf{try}(\tilde{s})\ \{\, I \,\}\ \mathbf{catch}\ \{\, J \,\} \searrow \begin{cases} (I', \tilde{t}) & \text{if } \tilde{s} \subseteq \tilde{t} \\ (\{\!\{J\}\!\} \mid I', \tilde{t} \cup \tilde{s}) & \text{otherwise} \end{cases}$$

$$\text{(G-MWrap)}\quad \{\!\{J\}\!\} \searrow (\{\!\{J\}\!\}, \emptyset)$$

$$\text{(G-MPar)}\quad I \searrow (I', \tilde{s}) \text{ and } J \searrow (J', \tilde{t}) \quad \Rightarrow \quad I \mid J \searrow (I' \mid J', \tilde{s} \cup \tilde{t})$$

$$\text{(G-MNil)}\quad I \searrow (\mathbf{0}, \emptyset) \quad \text{if } I \in \Big\{ \text{(inact),(init), (com), (cond), (recursion), (throw)} \Big\}$$

Table 2
Rules for meta reduction for the global calculus

then we just need to discard the current try-catch block. Otherwise, we need to take the parallel composition of the result I' and the wrapping of the handler J. The set $\tilde{t}$ is used for keeping track of nested refinements. (G-MWrap) leaves a wrapped global description unchanged while rule (G-MPar) applies meta reduction to the components of parallel composition and returns the composition of the

results together with the union of the session channel sets. Finally, (G-MNil) meta reduces any other choreography to the inaction $\mathbf{0}$.

The semantics rules reported in Table 1 also include (G-Clean). Using the information contained in the label of the reduction relation, this rule discards refined embedding try-catch blocks. Note that we apply meta reduction as I may also contain other try-catch blocks in parallel where an exception needs to be raised. (G-Res) removes session channels on the labels when there is a restriction. The context rule (G-Context) allows to compose try-catch blocks whenever the throw is not over refined channels (condition $\tilde{s} \notin \mathtt{fn}(C[-])$). (G-Rec) and (G-Par) are standard rules for recursion and parallel composition. Finally, (G-Struct) allows to use the standard structural congruence, defined as the least congruence relation $\equiv$ on I such that $\mid$ and $+$ are commutative monoids and it satisfies alpha-conversion and scope-opening $(\boldsymbol{\nu} s)\ I_1 \mid I_2 \equiv (\boldsymbol{\nu} s)\ (I_1 \mid I_2)$ for s not in I_2. The remaining rules in Table 1 are standard [4].

Example 2.3 *(Semantics of Financial Protocol with Broker)* We now show how the rules work in our second example. When $\mathtt{bad}(x)$ holds, we have that

$$(\boldsymbol{\nu} s)\ \ \mathbf{try}\ (s)\ \{\ \mathbf{if}\ \mathtt{bad}(x)\ \mathbf{then\ throw\ else}\ \ldots \text{lines 4-9} \ldots\ \}\ \mathbf{catch}\ \{\ J\ \}$$
$$\longrightarrow\longrightarrow$$
$$(\boldsymbol{\nu} s)\ \{\!\!\{\ J\ \}\!\!\}$$

where $J = \mathtt{Broker} \to \mathtt{Buyer} : s\langle \mathtt{reject}, \mathit{reason}, x\rangle\,.\,I$. In the other case, when Buyer wishes to accept the quote ($y < 100$ holds), we have:

$$\begin{array}{l} (\boldsymbol{\nu} s,t)\ \ \mathbf{try}\ (s)\ \{ \\ \quad \mathbf{try}\ (t,s)\ \{ \\ \quad\quad \mathbf{if}\ (y < 100)\ \mathbf{then\ throw} \\ \quad\quad \mathbf{else}\ \{\ldots \text{as line 4} \ldots\} \\ \quad \}\ \mathbf{catch}\ \{\ldots \text{as lines 8-9} \ldots\} \\ \}\ \mathbf{catch}\ \{\ J\ \} \end{array} \quad \longrightarrow\longrightarrow \quad \{\!\!\{\ \{\ldots \text{as lines 8-9} \ldots\}\ \}\!\!\}$$

where the first arrow corresponds to the reduction of the conditional to **throw** (for the sake of space we do not report the contractum term) with rules (G-IfT) and (G-Context); and the second arrow is then obtained by applying rules (G-Thr) and (G-Clean).

Types. The type discipline for the calculus is based on session types as in [3,4]. The grammar of types is as follows:

$$\alpha \ ::= \ \downarrow \Sigma_i \mathtt{op}_i(\theta_i)\,.\,\alpha_i \ \mid\ \uparrow \Sigma_i \mathtt{op}_i(\theta_i)\,.\,\alpha_i \ \mid\ \alpha\{\!\!\{\beta\}\!\!\} \ \mid\ \mathtt{end} \ \mid\ \mathbf{rec}\ \mathbf{t}\,.\,\alpha \ \mid\ \mathbf{t}$$

α is called *session type* while θ is a first-order value type (not a session type). The *try-catch type* $\alpha\{\!\!\{\beta\}\!\!\}$ is novel: it is the abstraction of a try-catch block where α

$$\text{(G-TInit)}\ \frac{\Gamma \vdash I \triangleright \prod_i t_i[A_i, B_i] : \alpha_i\{\!\{\beta_i\}\!\} \qquad \Gamma \vdash a@B_j : \alpha_j\{\!\{\beta_j\}\!\} \qquad \Gamma' \vdash J \triangleright \prod_i t_i[A_i, B_i] : \beta_i \qquad \Gamma' \subseteq \Gamma \qquad \mathsf{fv}(\Gamma') = \emptyset}{\Gamma \vdash A_j \longrightarrow B_j : a(t_j)[\tilde{t}, I, J] \triangleright \prod_{i \neq j} t_i[A_i, B_i] : \alpha_i\{\!\{\beta_i\}\!\}}$$

$$\text{(G-TCom)}\ \frac{\Gamma \vdash I \triangleright \Delta \cdot s[A, B] : \alpha_j \quad \Gamma \vdash e@A : \theta_j \quad \Gamma \vdash x@B : \theta_j \quad j \in K}{\Gamma \vdash A \to B : s\langle \mathsf{op}, e, x\rangle . I \triangleright \Delta \cdot s[A, B] : \downarrow \Sigma_{i \in K}(\theta_i) . \alpha_i}$$

$$\text{(G-TCom}^{-1}\text{)}\ \frac{\Gamma \vdash I \triangleright \Delta \cdot s[B, A] : \alpha_j \quad \Gamma \vdash e@A : \theta_j \quad \Gamma \vdash x@B : \theta_j \quad j \in K}{\Gamma \vdash A \to B : s\langle \mathsf{op}, e, x\rangle . I \triangleright \Delta \cdot s[B, A] : \uparrow \Sigma_{i \in K}(\theta_i) . \alpha_i}$$

$$\text{(G-TThr)}\ \frac{\mathsf{fv}(\Gamma) = \emptyset}{\Gamma \vdash \mathbf{throw} \triangleright \prod_i s_i[A_i, B_i] : \alpha_i} \qquad \text{(G-TRec)}\ \frac{\Gamma, X : \Delta \vdash I \triangleright \Delta}{\Gamma \vdash \mathbf{rec}\ X . I \triangleright \Delta}$$

$$\text{(G-TPar)}\ \frac{\Gamma \vdash I_i \triangleright \Delta_i}{\Gamma \vdash I_1 \mid I_2 \triangleright \Delta_1 \uplus \Delta_2} \qquad \text{(G-TVar)}\ \frac{\Gamma,\ X : \Delta \text{ well-formed}}{\Gamma,\ X : \Delta \vdash X \triangleright \Delta}$$

$$\text{(G-TIf)}\ \frac{\Gamma \vdash e@A : \mathsf{bool} \qquad \Gamma \vdash I_i \triangleright \Delta}{\Gamma \vdash \mathbf{if}\ e@A\ \mathbf{then}\ I_1\ \mathbf{else}\ I_2 \triangleright \Delta} \qquad \text{(G-TSum)}\ \frac{\Gamma \vdash I_i \triangleright \Delta}{\Gamma \vdash I_1 + I_2 \triangleright \Delta}$$

$$\text{(G-Zero)}\ \frac{\Gamma \text{ well-formed} \qquad s_i, t_j \text{ distinct}}{\Gamma \vdash \mathbf{0} \triangleright \Pi_i\, s_i[A_i, B_i] : \mathtt{end} \cdot \Pi_j\, t_j[A'_j, B'_j] : \mathtt{end}\{\!\{\beta_j\}\!\}}$$

Table 3
Typing Rules for the Global Calculus

denotes the type of a service in the default choreography while β the type in the exception handler. From now on in $\alpha\{\!\{\beta\}\!\}$, we stipulate that α and β do not contain any try-catch type.

The typing judgements have the form $\Gamma \vdash I \triangleright \Delta$ where Γ is the *service environment* and Δ is the *session environment*. They are formally defined as:

$$\begin{array}{llllllll} \textit{(Service)} & \Gamma & ::= & \emptyset & \mid & \Gamma,\, a@B : \alpha\{\!\{\beta\}\!\} & \mid & \Gamma,\, X : \Delta \quad \mid \quad \Gamma,\, x@A : \theta \\ \textit{(Session)} & \Delta & ::= & \emptyset & \mid & \Delta \cdot s[A, B] : \alpha & & \end{array}$$

In a service typing, $a@A : \alpha\{\!\{\beta\}\!\}$ says that a is located at A and offers a service interface $\alpha\{\!\{\beta\}\!\}$; $x@A : \theta$ says that a variable x located at A may store values of type θ; finally, $X : \Delta$ says that when the interaction recurs to X, it should have the typing Δ.

In session typing, the assignment $s[A, B] : \alpha$ says that a session channel s, belonging to a session between A and B, has the session type α when seen from the viewpoint of B.

The entailment relation $\vdash$ is defined by the rules reported in Table 3. (G-TInit) types session initiation. In the service environment, if channel a is located at B_j and has session type $\alpha_j\{\!\{\beta_j\}\!\}$ then we must make sure that channel t_j is used as β_j in the handler J and as $\alpha_j\{\!\{\beta_j\}\!\}$ in the default choreography (we carry β_j because of refinement). Note that β_j has no try-catch type nested: this follows from the assumption that in any type of the form $\alpha\{\!\{\beta\}\!\}$, none of its subterms is a try-catch type. As a consequence, (G-TInit) ensures that t_j is not refined in the handler.

Moreover, assumption $\mathsf{fv}(\Gamma') = \emptyset$, guarantees that there is no recursion variable occurring free in J. (G-Comm) and (G-Comm^{-1}) type in-session communication: they prefix the type of a session channel with an input and output type respectively. We need two typing rules because communication types can be expressed from the viewpoint of the sender (A) or the receiver (B). The operation $\uparrow \Sigma_{i\in K}(\theta_i)\,.\,\alpha_i$ creates optional several branches (like in standard session types) and it is read as $(\uparrow \Sigma_{i\in K}(\theta_i)\,.\,\alpha_i')\{\!\{\beta\}\!\}$ whenever $\alpha_j = \alpha_j'\{\!\{\beta\}\!\}$. In (G-TThr), term **throw** is typed such that any session channel s_i can have any type because this operation can interrupt any conversation. Rule (G-Par) uses disjoint union $\uplus$ in order to guarantee linearity [6]. Rule (G-Zero) allows to start typing (bottom-up) with inactive sessions (end for handlers) or inactive try-catch types ($\mathtt{end}\{\!\{\beta\}\!\}$ for default choreographies). The remaining rules are standard.

Example 2.4 *(Typing of Financial Protocol with Broker)* As an example, the types of services **chSeller** and **chBroker** are

$$(\mathbf{rec}\ \mathbf{t}.\uparrow \mathsf{update}(\mathtt{int})\,.\,\mathbf{t})\{\!\{\uparrow \mathsf{conf}(\mathtt{int})\,.\,\downarrow \mathsf{data}(\mathtt{int})\}\!\}$$

and

$$(\downarrow \mathsf{identify}(\mathtt{int})\,.\,\mathbf{rec}\ \mathbf{t}.\uparrow \mathsf{update}(\mathtt{int})\,.\,\mathbf{t})\{\!\{\ \uparrow \mathsf{conf}(\mathtt{int})\,.\,\downarrow \mathsf{data}(\mathtt{int}) + \uparrow \mathsf{reject}(\mathtt{string})\}\!\}$$

respectively. □

Types for Run-Time Choreography. In order to also type run-time terms and state subject reduction, we need two additional rules for typing try-catch blocks and wrapped choreographies. First, we enhance the type of session channels by redefining the session environment:

$$\Delta \quad ::= \quad \emptyset \quad | \quad \Delta \cdot s[A,B] :^n \alpha$$

Above, channel s is *protected* if $n = 1$, *unprotected* if $n = 0$. Session channels are protected whenever they occur inside a try-catch block or a wrap. Finally, the additional rules for typing the run-time syntax are:

$$\text{(G-TTry)}\ \frac{\begin{array}{c}\Gamma \vdash I \triangleright \prod_j t_j :^1 \alpha_j' \ \cdot\ \prod_i s_i :^{m_i} \alpha_i\{\!\{\beta_i\}\!\} \qquad m \in \{0,1\} \\ \Gamma' \vdash J \triangleright \prod_i \kappa_i :^0 \beta_i \qquad \Gamma' \subseteq \Gamma,\ \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash_A \mathbf{try}(\tilde{s})\ \{\,I\,\}\ \mathbf{catch}\ \{\,J\,\} \triangleright \prod_j t_j :^1 \alpha_j' \ \cdot\ \prod_i s_i :^1 \alpha_i\{\!\{\beta_i\}\!\}}$$

$$\text{(G-TWrap)}\ \frac{\Gamma \vdash J \triangleright \prod_j t_j :^1 \alpha_j\{\!\{\beta_j\}\!\} \cdot \prod_i s_i :^0 \beta_i'}{\Gamma \vdash_A \{\!\{J\}\!\} \triangleright \prod_j t_j :^1 \alpha_j\{\!\{\beta_j\}\!\} \cdot \prod_i s_i :^1 \alpha_i'\{\!\{\beta_i'\}\!\}}$$

$$\text{(G-TRes)}\ \frac{\Gamma \vdash I \triangleright \Delta \cdot s[A,B] : \alpha\{\!\{\beta\}\!\}}{\Gamma \vdash (\boldsymbol{\nu} s)\ I \triangleright \Delta}$$

Rule (G-TTry) types try-catch blocks and carries conditions similar to (G-TInit). The main difference lies on the typing of I where any session channel t_i not in $\tilde{s}$ must be protected i.e. there is a try-catch block on t_i. As a result, also channels in $\tilde{s}$ become protected. We impose this condition because we want to avoid to brutally terminate unprotected sessions, operation that would not have a real end-point implementation. In (G-TWrap), all those unprotected session channels become protected where their type becomes a try-catch type whose left-hand side is randomly chosen (α_i'). (G-TRes) is standard.

In the sequel, $\Gamma \vdash \sigma$ means that $\Gamma \vdash x@A : \theta$ for all $x@A$ in σ. The expected property of subject reduction holds like in the original version without exceptions.

Theorem 2.5 (Subject Reduction) *Assume $\Gamma \vdash \sigma$ and let I be derived from a non-run-time global description. Then $\Gamma \vdash I \triangleright \emptyset$ and $(\sigma, I) \to (\sigma', I')$ imply $\Gamma \vdash \sigma'$ and $\Gamma \vdash I' \triangleright \emptyset$.*

3 End-Point Interactional Exceptions

End-point interactional exceptions were first introduced in [5]. In this section, we briefly introduce the formalism enhanced with locations and such that branching/selection are embedded in standard communication (as in [4]).

Syntax. The syntax of processes (often called *programs*) and networks in the asynchronous end-point calculus with exceptions is reported below:

$P ::=$	$!c(\kappa)[P, Q]$	(service)	$\mid \overline{c}(\lambda)[\tilde{\kappa}, P, Q]$	(request)
	$\mid \kappa?\Sigma_i \mathsf{op}_i\langle x_i\rangle.P_i$	(input)	$\mid \kappa!\mathsf{op}\langle e\rangle \,.\, P$	(output)
	$\mid P \mid Q$	(par)	$\mid \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ P$	(cond)
	$\mid \mathbf{0}$	(inact)	$\mid P \oplus Q$	(choice)
	$\mid X$	(termVar)	$\mid \mathbf{rec}\ X \,.\, P$	(recursion)
	$\mid \mathbf{throw}$	(throw)		
$N ::=$	$A[\ P\ ]_\sigma \mid \quad N_1 \mid\mid N_2 \mid \quad \epsilon$			

where κ, λ denote polarised session channels s^+, s^-. The term (service) denotes a service c which, when invoked, will initiate a session κ with a default process P and a handler Q (to be used when an exception is thrown). Dually, term (request) represents the invocation of service c with session channel λ, default process P and handler Q. The vector $\tilde{\kappa}$ is used for refinement similarly to the choreography case. The remaining terms are standard. In the end-point calculus, we shall make the same assumptions about refinement and recursion done for the global calculus. Free session channels and term variables are also defined similarly.

A network N is the parallel composition of participants denoted by $A[\ P\ ]_\sigma$ where A is the participant's name and P is the process running at such location in a state σ. The latter is similar to the global case but it only maps variables (now local) to values.

$$\text{(E-MTry)}\quad P \searrow (P', \tilde{\kappa}') \;\Rightarrow\; \mathbf{try}(\tilde{\kappa})\ \{\, P \,\}\ \mathbf{catch}\ \{\, Q \,\} \searrow \begin{cases} (P', \tilde{\kappa}') & \text{if } \tilde{\kappa} \subseteq \tilde{\kappa}' \\ (\tilde{\kappa}\{\{Q\}\} \mid P', \tilde{\kappa}' \cup \tilde{\kappa}) & \text{otherwise} \end{cases}$$

$$\text{(E-MWrap)}\quad \tilde{\kappa}\{\{Q\}\} \searrow (\tilde{\kappa}\{\{Q\}\}, \emptyset)$$

$$\text{(E-MPar)}\quad P \searrow (P', \tilde{\kappa}_1) \text{ and } Q \searrow (Q', \tilde{\kappa}_2) \;\Rightarrow\; P \mid Q \searrow (P' \mid Q', \tilde{\kappa}_1 \cup \tilde{\kappa}_2)$$

$$\text{(E-MNil)}\quad R \searrow (\mathbf{0}, \emptyset) \quad \text{if } R \in \left\{ \begin{array}{l} \text{(zero),(request), (input), (output), (branch),} \\ \text{(select), (cond), (recursion), (throw)} \end{array} \right\}$$

Table 4
Rules for Meta Reduction for the End-Point Calculus

Semantics. In this subsection, we report the meta reduction and the reduction semantics for the end-point calculus with exceptions and locations (participants). Similarly to the global case, we need to extend the syntax with run-time terms:

$$\begin{array}{lllllll} P ::= & \ldots & | & \mathbf{try}(\tilde{s})\ \{\, P \,\}\ \mathbf{catch}\ \{\, P \,\} & | & \tilde{s}\{\{P\}\} \\ N ::= & \ldots & | & \kappa \hookrightarrow_{\phi} \overline{\kappa} : L & | & (\boldsymbol{\nu} s)\ N \\ L ::= & \epsilon & | & \mathsf{op}(v) :: L & | & \dagger :: L \end{array}$$

The try-catch block $\mathbf{try}(\tilde{s})\ \{\, P \,\}\ \mathbf{catch}\ \{\, P \,\}$ denotes a run-time (local) process running the default (local) process with a (local) handler ready to be executed if an exception is thrown (locally or remotely). Networks are extended with restriction and queues which are needed for implementing asynchronous messaging [7,5]. In a queue $\kappa \hookrightarrow_{\phi} \overline{\kappa} : L$, $\overline{\kappa}$ denotes the dual of κ i.e. $\overline{s^-} = s^+$ and $\overline{s^+} = s^-$ and L is a list denoting the content of the queue. The special message $\dagger$ denotes the throwing of an exception.

Similarly to the global calculus, also the end-point calculus features a meta reduction relation which is defined by the rules given in Table 4. (E-MTry), the most relevant rule, creates the wrapped handler only if $\tilde{\kappa}$ was not refined in P, otherwise it returns the meta reduced subterm P'.

As try-catch blocks can be nested, it is necessary to introduce evaluation contexts (at process level) defined by the following grammar:

$$E[-] \;::= \mathbf{try}(\tilde{\kappa})\ \{\, E[-] \,\}\ \mathbf{catch}\ \{\, Q \,\} \;\mid\; P \mid E[-] \;\mid\; \tilde{\kappa}\{\{E[-]\}\} \;\mid\; -$$

In the sequel, the function $\mathsf{wraps}(E[-])$ returns the set of polarised session names that occur under a wrap in $E[-]$ e.g. $\kappa_i \in \mathsf{wraps}(\tilde{\kappa}\{\{Q \mid -\}\})$.

The rules defining the reduction semantics of the end-point calculus with exceptions and locations are reported in Table 5. Rule (E-Init) initiates a session between two participants A and B running processes $!a(s^-)[P, Q]$ (service) and $\overline{a}(s^+)[\tilde{\kappa}, P', Q']$ (request) respectively. The reduction will result into the parallel composition of two try-catch blocks (corresponding to service and request) and two queues for asynchronous communication. Note that the rule requires the (service) term to be top-level (no context) because of the service channel principle (SCP)

$$\text{(E-Init)}\quad A[\,!a(s^-)[P,Q] \mid P'\,]_\sigma \parallel B[\,E[\overline{a}(s^+)[\tilde{\kappa},P',Q']]\,]_{\sigma'} \longrightarrow$$
$$(\boldsymbol{\nu} s)\left(\begin{array}{r} A[\,!a(s^-)[P,Q] \mid \mathbf{try}(s^-)\,\{\,P\,\}\,\mathbf{catch}\,\{\,Q\,\} \mid P'\,]_\sigma \parallel s^- \hookrightarrow_0 s^+ : \epsilon \parallel \\ B[\,E[\mathbf{try}(\tilde{\kappa})\,\{\,P'\,\}\,\mathbf{catch}\,\{\,Q'\,\}]\,]_{\sigma'} \parallel s^+ \hookrightarrow_0 s^- : \epsilon \end{array}\right)$$

$$\text{(E-Out)}\quad \frac{\sigma \vdash e \Downarrow v}{A[\,E[\kappa!\mathsf{op}\langle e\rangle\,.\,P]\,]_\sigma \parallel \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow A[\,E[P]\,]_\sigma \parallel \kappa \hookrightarrow_\phi \overline{\kappa} : (\mathsf{op}(v) :: L)}$$

$$\text{(E-In)}\quad \frac{\kappa \in \mathsf{wraps}(C) \text{ iff } \phi = 1 \qquad L = L' :: \mathsf{op}(v)}{A[\,E[\kappa?\Sigma_i \mathsf{op}_i\langle x_i\rangle.P_i]\,]_\sigma \parallel \overline{\kappa} \hookrightarrow_\phi \kappa : L \longrightarrow A[\,E[P\{v/x\}]\,]_\sigma \parallel \overline{\kappa} \hookrightarrow_0 \kappa : L'}$$

$$\text{(E-Thr)}\quad \frac{P' \searrow (R,\tilde{\kappa}) \qquad \mathsf{outerTry}(P',\kappa_1) \qquad P' = E'[\mathbf{try}(\tilde{\kappa})\,\{\,\mathbf{throw} \mid P\,\}\,\mathbf{catch}\,\{\,Q\,\}]}{A[\,E[P']\,]_\sigma \parallel \Pi_{\kappa\in\tilde{\kappa}'}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow A[\,E[R]\,]_\sigma \parallel \Pi_{\kappa\in\tilde{\kappa}'}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)}$$

$$\text{(E-RThr)}\quad \frac{P \searrow (R,\tilde{\kappa}) \qquad \mathsf{outerTry}(P,\kappa_j)}{\begin{array}{l} A[\,E[P]\,]_\sigma \parallel \overline{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \parallel \Pi_{\kappa\in\tilde{\kappa}}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow \\ A[\,E[R]\,]_\sigma \parallel \overline{\kappa}_j \hookrightarrow_1 \kappa_j : L \parallel \Pi_{\kappa\in\tilde{\kappa}}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa) \end{array}}$$

$$\text{(E-WVal)}\quad A[\,E[\tilde{\kappa}\{\{Q\}\}]\,]_\sigma \parallel \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \mathsf{op}(v)) \longrightarrow A[\,E[\tilde{\kappa}\{\{Q\}\}]\,]_\sigma \parallel \overline{\kappa}_i \hookrightarrow_0 \kappa_i : L$$

$$\text{(E-WThr)}\quad A[\,E[\tilde{\kappa}\{\{Q\}\}]\,]_\sigma \parallel \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \dagger) \longrightarrow A[\,E[\tilde{\kappa}\{\{Q\}\}]\,]_\sigma \parallel \overline{\kappa}_i \hookrightarrow_1 \kappa_i : L$$

$$\text{(E-IfTT)}\quad \sigma \vdash e \Downarrow \mathsf{tt} \ \Rightarrow\ A[\,E[\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q]\,]_\sigma \longrightarrow A[\,E[P]\,]_\sigma$$

$$\text{(E-IfFF)}\quad \sigma \vdash e \Downarrow \mathsf{ff} \ \Rightarrow\ A[\,E[\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q]\,]_\sigma \longrightarrow A[\,E[Q]\,]_\sigma$$

$$\text{(E-Struct)}\quad \frac{N_1 \equiv N_1' \quad N_1 \longrightarrow N_2 \quad N_2 \equiv N_2'}{N_1' \longrightarrow N_2'} \qquad \text{(E-Res)}\quad \frac{N \longrightarrow N'}{(\boldsymbol{\nu} s)\ N \longrightarrow (\boldsymbol{\nu} s)\ N'}$$

$$\text{(E-Sum)}\quad A[\,E[P \oplus Q]\,]_\sigma \longrightarrow A[\,E[P]\,]_\sigma \qquad \text{(E-Par)}\quad \frac{N \longrightarrow N''}{N \parallel N' \longrightarrow N'' \parallel N'}$$

$$\text{(E-Rec)}\quad \frac{A[\,E[P\{\mathbf{rec}\ X\,.\,P/X\}]\,]_\sigma \longrightarrow A[\,E[P']\,]_\sigma}{A[\,E[\mathbf{rec}\ X\,.\,P]\,]_\sigma \longrightarrow A[\,E[P']\,]_\sigma}$$

Table 5
Rules for the Reduction Semantics of the End-Point Calculus with Exceptions

[4] (service channels can be shared and invoked repeatedly). (E-Out) implements the asynchronous output consisting in putting v, the evaluation of the expression e in the state σ ($\sigma \vdash e \Downarrow v$), into the queue together with the operation op (used for selecting branches). Rule (E-In) defines the dual operation of consuming an element $\mathsf{op}_j(v)$ from the queue and selecting one of the branches op_i. Note that if the input term is in a try-catch block (over κ) then the queue level must be 0 whereas a queue level equal to 1 requires that the input term is in a wrap (over κ). This ensures that if an exception occurs only those messages sent after the throw are read. Comparing to the original version in [5], we do not allow queues to enter try-catch blocks (in here we have locations). Local throwing of exceptions is defined by (E-Thr). Together with (E-RThr), this rule uses the end-point meta reduction relation $\searrow$ and $\mathsf{outerTry}$. The predicate $\mathsf{outerTry}(P,\kappa)$ is used for finding the outermost try-catch block which is connected to κ by refinement. Formally,

$$\mathsf{outerTry}(P,\kappa) = \begin{cases} \mathsf{true} & \text{if } P = \mathbf{try}(\kappa)\,\{P'\}\,\mathbf{catch}\,\{\,Q\,\} \text{ or } (P = P_{\mathsf{n}} \ \wedge\ \kappa,\kappa' \in \tilde{\kappa}) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

where $P_{\mathsf{n}} = \mathbf{try}(\kappa')\ \{\, E[\mathbf{try}(\tilde{\kappa})\ \{\, P'' \,\}\ \mathbf{catch}\ \{\, Q' \,\}]\,\}\ \mathbf{catch}\ \{\, Q \,\}$. In the following example, we have $\mathsf{outerTry}(P, \kappa)$ and $\neg\mathsf{outerTry}(P', \kappa)$:

$$P = \mathbf{try}(\kappa')\ \{\, \mathbf{try}(\kappa, \kappa')\ \{\, P \,\}\ \mathbf{catch}\ \{\, Q \,\}\,\}\ \mathbf{catch}\ \{\, Q' \,\};$$
$$P' = \mathbf{try}(\kappa, \kappa')\ \{\, \mathbf{try}(\kappa, \kappa', \kappa'')\ \{\, P \,\}\ \mathbf{catch}\ \{\, Q \,\}\,\}\ \mathbf{catch}\ \{\, Q' \,\}.$$

Using the above definition assumes the existence of a try-catch block over a single κ' which is refined in another nested try-catch block where it is coupled with κ. This is always true whenever we consider processes derived from programs i.e. syntax in pag. 11.
The predicate $\mathsf{outerTry}$ is necessary because the throwing of an exception can happen nested in a try-catch block which is a refinement of an outer one. Like in the financial protocol with broker, the latter needs to be discarded. Hence, we need to apply meta reduction to the outermost block.

After applying meta reduction, the special message † is put into the queue. Simmetricaly, Rule (E-RThr) is applicable when † is on the queue hence meaning the presence of a remote throw. In this case, the exception is propagated to the other channels and the queue level where the exception was received is now set to 1 so that new messages, asynchronously put after †, are only delivered to the wrapped handler. (E-WVal) and (E-WThr) take care of removing all those messages sent before an exception was thrown and setting the queue level to 1 when removing † respectively. The rest of the rules is standard. Rule (E-Struct) uses a standard structural congruence $\equiv$. The relation is defined as the least congruence on processes such that $(\equiv, \oplus)$, $(\equiv, +)$ and $(\equiv, |)$ are commutative monoids. $\equiv$ is then extended to networks such that $(|, \epsilon)$ is a commutative monoid and

$$(\boldsymbol{\nu} s)\ \epsilon \equiv \epsilon$$
$$A[\, P \,]_\sigma \equiv A[\, Q \,]_\sigma \quad (\text{if } P \equiv Q)$$
$$(\boldsymbol{\nu} s_1)\ (\boldsymbol{\nu} s_2)\ M \equiv (\boldsymbol{\nu} s_2)\ (\boldsymbol{\nu} s_1)\ M$$
$$(\boldsymbol{\nu} s)\ M \mid N \equiv (\boldsymbol{\nu} s)\ (M \mid N) \qquad \text{for } s \notin \mathsf{fn}(N)$$

Types. The syntax of session types is identical to the global case. Typing judgements for networks, processes and expressions have the forms $\Gamma \vdash N \triangleright \Delta$, $\Gamma \vdash_A P \triangleright \Delta$ and $\Gamma \vdash e : \theta$ respectively where Γ is a *service typing*, which typically maps service channels to service types and Δ is a *session typing* which typically maps session channels to session types. For $n \in \{0, 1\}$ and $\rho \in \{p, u\}$, typings are defined as

$$\textit{(Session Typing)} \quad \Delta ::= \emptyset \mid \Delta,\, \kappa :^n_\rho \alpha \mid \Delta,\, (\kappa, \overline{\kappa}) : \alpha \mid \Delta,\, (\kappa, \overline{\kappa}) : \bot$$
$$\textit{(Service Typing)} \quad \Gamma ::= \emptyset \mid \Gamma,\, c : \langle \alpha \{\!\!\{ \beta \}\!\!\} \rangle \mid c : \mathtt{bool} \mid \Gamma,\, X : \Delta$$

In session typings, $\kappa :^n_\rho \alpha$ says that: *at a polarised session channel κ, there is a session of type α*. The natural number n is equal to 1 if there is a wrap on κ, 0 otherwise. A session channel with respect to its type is *unprotected* if $\rho = u$ (no try-catch nor wrap on κ occurs) and *protected* if $\rho = p$ (there is a try-catch or a wrap on κ). This is needed in the try-catch and wrap typing as well as in the merging with the queue types $(\kappa, \overline{\kappa}) : \alpha$ and $(\kappa, \overline{\kappa}) : \bot$ used for typing a queue from κ to $\overline{\kappa}$ (the

$$\text{(E-TReq)}\ \frac{\begin{array}{l}\Gamma \vdash_A P \triangleright \prod_i \kappa_i :^0_u \overline{\alpha}_i\{\!\{\overline{\beta}_i\}\!\} \\ \Gamma' \vdash_A Q \triangleright \prod_i \kappa_i :^0_u \overline{\beta}_i \qquad s^+ = \kappa_j \\ \Gamma \vdash c : \langle \alpha_j\{\!\{\beta_j\}\!\}\rangle \qquad \Gamma' \subseteq \Gamma,\ \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash_A \overline{c}(s^+)[\tilde{\kappa}, P, Q] \triangleright \prod_{i\neq j} \kappa_i :^0_u \overline{\alpha}_i\{\!\{\overline{\beta}_i\}\!\}}$$

$$\text{(E-TServ)}\ \frac{\begin{array}{l}\Gamma \vdash_A P \triangleright s^- :^0_u \alpha\{\!\{\beta\}\!\} \\ \Gamma \vdash_A Q \triangleright s^- :^0_u \beta \qquad \mathsf{fv}(\Gamma) = \emptyset\end{array}}{\Gamma,\ a : \langle \alpha\{\!\{\beta\}\!\}\rangle \vdash_A !a(s^-)[P, Q] \triangleright \emptyset}$$

$$\text{(E-Part)}\ \frac{\Gamma \vdash_A P \triangleright \Delta \qquad \Gamma \vdash \sigma @ A}{\Gamma \vdash A[\, P \,]_\sigma \triangleright \Delta}$$

$$\text{(E-Par)}\ \frac{\Gamma \vdash N_1 \triangleright \Delta_1 \qquad \Gamma \vdash N_2 \triangleright \Delta_2}{\Gamma \vdash N_1 \mid\mid N_2 \triangleright \Delta_1 \odot \Delta_2}$$

Table 6
Selected typing rules for the end-point calculus

type of a queue is composed with the type of a process in which case the queue's type becomes $\bot$).

In the service typing, c either has type $\alpha\{\!\{\beta\}\!\}$ (a service using a session channel with default behaviour of type α and with a handler of type β) or an atomic type such as `bool`.

The typing system for programs is identical to the one given in [5]. In Table 6 we report the typing rules for process terms (request) and (init), and the ones for networks $A[\, P \,]_\sigma$ and $N \mid\mid N'$. Rule (E-TReq) types a service request. The typing system makes sure that $\tilde{\kappa}$ is not wrapped in P and Q as well as it is unprotected in the handler Q (we do not allow re-try in the catch part). (E-TServ) is its dual rule. The rules for network are identical to the ones found in [4].

Subjected reduction also holds for this version of the end-point calculus with exceptions and locations. In the following, $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Theorem 3.1 (Subject Reduction) *Let N be a program such that $\Gamma \vdash N \triangleright \emptyset$. If $N \longrightarrow^* N'$ then $\Gamma \vdash N' \triangleright \emptyset$.*

4 On the End-Point Projection

In this section, we discuss the main issues regarding the procedure of generating end-point code from a given choreography known as *end-point projection* (EPP). EPP is not always feasible as the global calculus allows to write global interactions without a reasonable implementation. Enforcing some extra conditions on the global calculus allows to design systems which enjoy a sound EPP. In the sequel, we analyse, through some examples, the three algorithmically-checkable conditions introduced in [4].

Connectedness. Connectedness is about causality of interactions. The choreography $A \to B : s\langle \mathsf{op}, e, x\rangle \,.\, C \to D : t\langle \mathsf{op}', e', y\rangle$ describes a system where participant A is sending a message to participant B and, once this communication has happened, participant C will send a message to participant D. If we think of a possible implementation of the described system we need to make sure that we implement a notification from A or B to C or D which will trigger the second communication. In order to avoid this, we require that sequential interactions are connected i.e. at least one participant between A and B also occurs in the second interaction.

In [4], a very strong notion of connectedness is used (in the above example, B must be equal to C). We believe that the notion should be relaxed to a softer form, at least allowing choreographies such as the one below

$$\begin{array}{l} A \to B : b(s)[\tilde{t}, \\ \quad B \to A : s\langle \mathsf{op}_1, e_1, x\rangle \,.\, B \to A : s\langle \mathsf{op}_2, e_2, y\rangle, \\ \quad A \to B : s\langle \mathsf{op}_3, e_3, z\rangle] \end{array}$$

Above, the default choreography describes a system where participant B sends two messages, one immediately after the other, without waiting for an acknowledgement from A. This is a common scenario in communication-centred programming that, we believe, should also be allowed. As another example, also consider the financial protocol shown in the previous sections (also without Broker) where Seller repeatedly sends a quote.

In general, we want a connectedness condition requiring that either i) $B = C$ or ii) $A = C$ and $B = D$ [5] in $A \to B : s\langle \mathsf{op}, e, x\rangle \,.\, C \to D : t\langle \mathsf{op}', e', y\rangle$.

Well-Threadedness. Well-threadedness is also a causality condition but more subtle than connectedness. Rather than being related to which participants are performing an interaction, well-threadedness also considers how sessions are used. As an example, consider the following global description:

$$A \to B : b(s)[s, B \to C : c(r)[r, \underbrace{C \to A : a(u)[u, \overbrace{A \to B : s\langle \mathsf{op}, v, x\rangle}^{2}, J_1]}_{1}, J_2], J_3]$$

Regardless of J_i, the description above is unrealisable at end-points. In fact, the first action tells that a thread (local process) in A invokes B. Such a thread then becomes inactive in part 1 where service c at A is invoked. In part 2, A communicates with B via s, the session channel created in the initial action. If we try to write the behaviour above as an end-point calculus term, taking into account the service channel principle mentioned in the previous section, we have (for some σ_A, σ_B and σ_C):

$$\begin{array}{lllll} A[& \overline{b}(s)[s, s!\mathsf{op}\langle v\rangle, Q_1] & \mid & !a(t)[\mathbf{0}, Q_2] &]_{\sigma_A} \quad \mid \\ B[& !b(s)[& \overline{c}(u)[u, & s?\mathsf{op}(x) \,.\, P', Q_4] & , Q_3] \;]_{\sigma_B} \quad \mid \\ C[& !c(t)[& \overline{a}(u)[u, \mathbf{0}, Q_5], Q_6] & &]_{\sigma_C} \end{array}$$

At A, the process on the left invokes b and sends v with operation op in the same session, while the one on the right is a service and by SCP this channel should constantly be ready to receive invocations. Note that $s!\mathsf{op}\langle v\rangle$ cannot be located under a, as it belongs to a session s. When the three processes interact, first, A invokes b, then B invokes c following the original global scenario. However, at this point, the action $s?\mathsf{op}(x)$ is free to react with its dual action $s!\mathsf{op}\langle v\rangle$ thus violating the sequencing in the global description. Whenever a global description is free from such false dependency, we say it is *well-threaded* [4].

[5] This differs from the definition given in [4] where we only consider condition i)

Well-threadedness is based on thread annotation of global descriptions. Threads are denoted by identifiers denoting which actions belong to the same processes. For instance, in the example above, $\overline{b}(s)[s, s!\mathsf{op}\langle v\rangle, Q_1]$ and $!a(t)[\mathbf{0}, Q_2]$ denote two different threads, both running at the same location A.

In general, we extend the syntax of the global calculus to annotated interactions. As for the new constructs introduced in this paper, we would have:

$$\mathcal{A} ::= \ \ldots \ \mid \ A^{\tau_1} \to B^{\tau_2} : b(s)[\tilde{t}, \mathcal{A}, \mathcal{A}'] \quad \mid \ \mathbf{try}(\tilde{t}) \ \{\, \mathcal{A} \,\} \ \mathbf{catch} \ \{\, \mathcal{A}' \,\} \ \mid \ \mathbf{throw}^{A,\tau}$$

where each τ_i is a natural number. We call $\tau, \tau', \ldots$ occurring in an annotated interaction, *threads*.

Example 4.1 Below, we show a possible but not unique annotation of the financial protocol with broker.

1. $\mathtt{Buyer}^1 \to \mathtt{Broker}^2$: $\mathbf{chBroker}(s)$ $[s,$
2. $\mathtt{Buyer}^1 \to \mathtt{Broker}^2 : s\langle \mathtt{identify}, id, x\rangle$.
3. **if** $\mathtt{bad}(x)@\mathtt{Broker}^2$ **then throw**$^{\mathtt{Broker},\,2}$
4. **else** $\mathtt{Broker}^2 \to \mathtt{Seller}^3$: $\mathbf{chSeller}(t)[(s,t),$
5. **rec** $X_3^{\mathtt{Seller}}$. $\mathtt{Seller}^3 \to \mathtt{Broker}^2 : t\langle \mathtt{update}, quote, y\rangle$.
6. $\mathtt{Broker}^2 \to \mathtt{Buyer}^1 : s\langle \mathtt{update}, y + 10\%, y\rangle$.
7. **if** $(y < 100)@\mathtt{Buyer}^1$ **then throw**$^{\mathtt{Buyer},\,1}$ **else** X_1^A, } default
8. $\mathtt{Seller}^3 \to \mathtt{Broker}^2 : t\langle \mathtt{conf}, cnum, x\rangle$.
9. $\mathtt{Broker}^2 \to \mathtt{Buyer}^1 : s\langle \mathtt{conf}, x, x\rangle$.
10. $\mathtt{Buyer}^1 \to \mathtt{Broker}^2 : s\langle \mathtt{data}, credit, x\rangle$.
11. $\mathtt{Broker}^2 \to \mathtt{Seller}^3 : t\langle \mathtt{data}, x, x\rangle]$, } handler } default
12. $\mathtt{Broker}^2 \to \mathtt{Buyer}^1 : s\langle \mathtt{reject}, reason, x\rangle$. $\mathcal{A}$] } handler

□

Thread Projection and Coherence. In this subsection, we discuss, together with the definition of coherence, the notion of thread projection i.e. given an annotated global description I, how can we generate the end-point behaviour associated to each of its threads.

As an example, the choreography $A^1 \to B^2 : s\langle \mathsf{op}, e, x\rangle$ contains two threads (thread 1 and 2). The projection of thread 1 is an output of expression e on session channel s with operation op i.e. $s!\mathsf{op}\langle e\rangle$. On the other hand, the projection of thread 2 is $s?\mathsf{op}(x)$.

Unfortunately, when adding the exception mechanism to the global calculus, projection is not so straightforward when it comes to the thread projection of the

session initiation $A^1 \longrightarrow B^2 : b(s)[\tilde{t}, I, J]$. In fact, such an operation also contains the refinement $\tilde{t}$. Now, consider the following possible projection, for $\tilde{\kappa}$ being a polarised version of $\tilde{t}$:

$$\begin{array}{ll} \text{(thread 1)} & \overline{b}(s^+)[\tilde{\kappa}, \mathsf{projection}(I,1), \mathsf{projection}(I,1)] \\ \text{(thread 2)} & !b(s^-)[\mathsf{projection}(I,2), \mathsf{projection}(I,2)] \end{array}$$

Is the above projection correct? Unfortunately, global interaction I could further refine $\tilde{t}$ in a service invocation from B to a third participant and the thread above would not be ready to deal with a refined handler. A similar argument could be done for thread 2. In order to further clarify, let us consider the following annotated choreography:

$$\begin{array}{ll} 1. & A^1 \to B^2 : b(s)[s, \\ 2. & \quad B^2 \to C^3 : c(t)[(t, s), \\ 3. & \qquad C^3 \to B^2 : t\langle \mathsf{op}_1, e_1, x_1\rangle \,.\, B^2 \to A^1 : s\langle \mathsf{op}_2, e_2, x_2\rangle, \\ 4. & \qquad C^3 \to B^2 : t\langle \mathsf{op}_3, e_3, x_3\rangle \,.\, B^2 \to A^1 : s\langle \mathsf{op}_4, e_4, x_4\rangle \ ], \\ 5. & \quad B^2 \to A^1 : s\langle \mathsf{op}_5, e_5, x_5\rangle \ ] \end{array}$$

When we project thread 1, we must take into consideration that, after the refinement in line 2, such a thread is involved in another handler in the line 4 besides the outermost handler in line 5. Therefore, we must make sure that the thread projection of 1 provides a single handler ready to input with both operations op_4 and op_5 (lines 4 and 5). The solution is to merge the outermost handler in the fifth line with any other nested handler involving thread 1. In [4], thread merging is a partial operation between threads that allows differences in branches which do not overlap. If two end-point behaviours are mergeable, we can merge them and obtain a single process which simulates both of the two behaviours, by combining missing branches from the both. For instance, in the above example the projection of thread 1 would result into

$$\overline{b}(s^+)[s^+, \qquad s^+?\mathsf{op}_2(x_2), \qquad \mathsf{merge}(s^+?\mathsf{op}_5(x_5),\ s^+?\mathsf{op}_4(x_4))\ ]$$

where $\mathsf{merge}(s^+?\mathsf{op}_5(x_5),\ s^+?\mathsf{op}_4(x_4)) = s^+?\{\mathsf{op}_5(x_5) + \mathsf{op}_4(x_4))$.

As mentioned above, merging is not always feasible e.g. in the example above $\mathsf{op}_5 = \mathsf{op}_4$ would not be allowed. In general, a well-threaded, consistently annotated interaction is *coherent* if for each of its threads, the thread projection is well-defined.

We conclude this section by showing the thread projection of the Financial Protocol with Broker.

Example 4.2 *(Thread Projection of Financial Protocol with Broker).* Hereby, we apply the projection function to generate the end-point code for the three threads reported in the previous section annotating the financial protocol with broker. For

thread 1, we have that $\mathsf{TP}(Protocol, 1)$ is:

$$\overline{\mathbf{chBroker}}(s^+)[s^+,$$
$$s^+!\mathtt{identify}\langle id\rangle\,.\,\mathbf{rec}\ X\,.\,s^+?\mathtt{update}(y)\,.\,\mathbf{if}\ (y < 100)\ \mathbf{then\ throw\ else}\ X,$$
$$\mathsf{merge}(s^+?\mathsf{conf}(x)\,.\,s^+!\mathsf{data}\langle credit\rangle,\ s^+?\mathsf{reject}(x)\,.\,P)\ ]$$

where the merging $\mathsf{merge}(s^+?\mathsf{conf}(x)\,.\,s^+!\mathsf{data}\langle credit\rangle,\ s^+?\mathsf{reject}(x)\,.\,P)$ is equal to $s^+?(\mathsf{conf}(x)\,.\,s^+!\mathsf{data}\langle credit\rangle + \mathsf{reject}(x)\,.\,P)$ where P is the thread projection of 1 in $\mathcal{A}$. The projection of thread 2, $\mathsf{TP}(Protocol, 2)$ is

$$*\mathbf{chBroker}(s^-)[$$
$$s^-?\mathsf{identify}(x)\,.\,\mathbf{if}\ \mathtt{bad}(x)\ \mathbf{then\ throw}$$
$$\mathbf{else}\ \overline{\mathbf{chSeller}}\,(t^+)[(s^-, t^+),$$
$$\mathbf{rec}\ X\,.\ t^+?\mathtt{update}(y)\,.\,s^-!\mathtt{update}\langle y + 10\%\rangle\,.\,X \quad \}\ \text{default}$$
$$t^+?\mathtt{conf}(x)\,.\,s^-!\mathtt{conf}\langle x\rangle\,.\,s^-?\mathtt{data}(x)\,.\,t^+!\mathtt{data}\langle x\rangle], \quad \}\ \text{handler}$$
$$\}\ \text{default}$$
$$s^-!\mathtt{reject}\langle reason\rangle\,.\,\mathcal{A}\ ] \quad \}\ \text{handler}$$

Finally, thread 3 has the following projection:

$$*\mathbf{chSeller}(t^-)[\ \mathbf{rec}\ X\,.\,t^-!\mathsf{update}\langle quote\rangle\,.\,X,\ t^-!\mathsf{conf}\langle cnum\rangle\,.\,t^-?\mathsf{data}(x)\ ]$$

Note that thread projection is not participant projection. In fact, each participant may contain more than one single service (unlike this example meant to focus on the exception aspect of choreography).

5 Conclusions

We have introduced the notion of exception for choreography. In particular, we have extended the syntax of the global calculus [3,4] with the exception mechanism and given its formal semantics. The aim of this work was to show how exceptions can be used at choreography level and, with examples, how they can be mapped to end-points. In the global calculus, exceptions are a simple form of transferring execution to a different choreography. But, together with a sound end-point projection, choreography becomes a powerful tool for designing end-point behaviour where the raising of an exception will transfer the execution of all end-points to an exception handling interaction.

Future Work. The global calculus with exceptions introduced in this work is still limited to some basic operators and only includes values in messages. One point that needs further investigation is to allow the passing of service and session channels although this problem should also be carefully studied at end-point level.

The end-point projection has only been analysed through examples but it has been revealed that it does not follow directly from the work done in [4]. However, the

three conditions for guaranteeing end-point projection need to be formally defined and a theorem stating the correctness of the EPP must be proved. In particular, we need to understand how to formally define the thread projection of run-time terms such as try-catch blocks and queues.

Related Work. The global calculus was first introduced in [3]. The version introduced in this work is an extension of the previous one in the fact that it also models interactional exceptions. The end-point projection has already been studied for choreography in general (e.g. [4,2]). The ideas contained in this work follow strictly the ones in [4] but propose further relaxations which have not been considered before. Exceptions in general have been studied for many programming languages including communication-based ones: in concurrent programming of distributed objects, [11] proposes an algorithm to resolve multiple kinds of exceptions among concurrently running objects is proposed; [1] introduces a model for long-running transactions which treats failures by restoring the initial state and firing a compensation process; the calculus for web services called COWS [8] provides an operation to kill processes except those protected by wraps similar to our exception mechanism; [9] introduces a calculus for web services by extending the π-calculus with exceptions. None of the aforementioned works relays on interactional exceptions, where conversations among peers together move to different ones when an exceptions is thrown.

References

[1] L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS'03*, LNCS, pages 124–138. Springer, 2003.

[2] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration conformance for system design. In *Coodination'06*, volume 4038 of *LNCS*, pages 63–81, 2006.

[3] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. In *2nd Workshop on Developments in Computational Models (DCM)*, ENTCS, 2006.

[4] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.

[5] M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions for session types. In *19th International Conference on Concurrency Theory (Concur'08)*, LNCS, pages 402–417. Springer, 2008.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *7th European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[7] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *35th Symposium on Principles of Programming Languages (POPL'08)*, pages 273–284. ACM, 2008.

[8] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.

[9] H. Vieira, L. Caires, and J. Seco. The conversation calculus: A model of service oriented computation. In *17th European Symposium on Programming (ESOP'08)*, volume 4421 of *LNCS*, pages 269–283. Springer, 2008.

[10] Web Services Choreography Working Group. http://www.w3.org/2002/ws/chor/.

[11] J. Xu, A. B. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.