Global Escape in Multiparty Sessions*

Sara Capecchi[1], Elena Giachino[2], and Nobuko Yoshida[3]

1 **Dipartimento di Informatica, Università di Torino**
Corso Svizzera 185, Torino, Italy
`capecchi@di.unito.it`

2 **Focus Reasearch Team, Università di Bologna/INRIA**
Mura Anteo Zamboni 7, Bologna , Italy
`giachino@cs.unibo.it`

2 **Imperial College London**
South Kensington Campus, London SW7 2AZ, Great Britain
`yoshida@doc.ic.ac.uk`

Abstract

This paper proposes a global escape mechanism which can handle unexpected or unwanted conditions changing the default execution of distributed communicational flows, preserving compatibility of the multiparty conversations. Our escape is realised by a collection of asynchronous local exceptions which can be thrown at any stage of the communication and to any subsets of participants in a multiparty session. This flexibility enables to model complex exceptions such as criss-crossing global interactions and fault tolerance for distributed cooperating threads. Guided by multiparty session types, our semantics automatically provides an efficient termination algorithm for global escapes with low complexity of exception messages.

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2010.338

1 Introduction

In multiparty distributed conversations, a frequent communication pattern is the one that provides that some unexpected condition may arise forcing the conversation to abort or to take suitable measures for handling the situation, usually by moving to another stage. Such a *global escape* may be not a computational error but rather a controlled, structured interruption requested by some participant. This paper proposes a structured global escape mechanism based on multiparty session types, which can control multiple interruptions efficiently, and guarantee deadlock-freedom without additional overheads. Our main focus is on *interactional exceptions*, which perform not only local management of the interrupted flows but also explicitly coordinate a set of collaborating and communicating peers. Interactional exceptions based on multiparty sessions provide the following contributions (in which relies the novelty of our approach w.r.t. [4, 6]):

- an extension of multiparty sessions [9] to flexible exception handling: we allow asynchronous escape at any desired point of a conversation, including nested exceptions;
- a flexible exceptions representation for modelling both "light" exceptions, representing a control flow mechanism rather than an error (such as time-outs), and "heavy" exceptions such as component or system crashes;
- a compositional model where nested exception contexts are not a refinement of the outer ones but inner isolated contexts involving only a subset of participants who can handle an unexpected situation without affecting the unrelated communications among other cooperating peers;

* This work has been partially supported by MIUR Projects DISCO (Distribution, Interaction, Specification, Composition for Object Systems), EPSRC EP/G015635/1 and EPSRC EP/F003757/1.

© Sara Capecchi, Elena Giachino and Nobuko Yoshida;
licensed under Creative Commons License NC-ND

IARCS Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010).
Editors: Kamal Lodaya, Meena Mahajan; pp. 338–351

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- exception signals modelled as natural linguistic constructs with a corresponding behaviours;
- applications to large scale protocols among multiple peers, automatically offering communication safety and deadlock-freedom. We apply our theory for well-known distributed protocols for exception handling and resolutions (CAs) [13], and prove our method offers the lower complexity w.r.t. a number of message exchanges than the previously studied algorithms.

As in [4] our extension is *consistent* (since despite synchrony and nesting of exceptions, communications in default and exception handling conversations do not mix) and *safe* (since linearity of communications inside sessions and absence of communication mismatch are enforced carrying out fundamental properties of session types). We ensure these properties using: (i) an asynchronous linguistic construct for exceptions signalling; (ii) multi-level queues: the different levels are used to avoid the mix of messages belonging to standard conversations and exception handling ones, which belong to different nesting levels; (iii) a type discipline based on the known technique of defining global types that describe the whole conversation behaviour, and projected end-point types classifying single peers behaviours.

The paper is organised as follows: in Section 2 we describe the syntax of our calculus and present an example. In Sections 3, 4 we present semantics, typing and properties of our extension; Section 5 shows how a known distributed object model can be encoded with our calculus. Finally Section 6 closes the paper. Detailed definitions, proofs and more examples can be found in [3].

2 Multiparty Session Processes with Exceptions

We introduce the syntax of processes using the π−calculus with multiparty sessions [9]. The syntax is given in Figure 1, where we use P and Q to range over process names, s over private channels, r over indexed private channels (of the form s^{φ}), a over public channels, v over values, e over expressions, x,y,z over variables, X, Y over term variables and l, l_i over labels. We adopt the notation $\tilde{s}$ as a shorthand for $s_1, \ldots, s_n$.

The session connection is performed by a multicast request $\overline{a}[2..n](\tilde{s}).P$ (over the public channel a, specifying the number n of participants invited) and n accept operations $a[\mathtt{p}](\tilde{s}).P$ (over the same channel a). In both cases $\tilde{s}$ are the private channels that will be used in the continuation. A process engaged in a session can perform an output action $r!\langle e\rangle$, sending on r the evaluation of the expression e, an input action $r?(x).P$, receiving a value on r, bound by x in P. More than one labelled behaviour may be offered on the channel r ($r \triangleright \{l_i : P_i\}_{i\in I}$), so that it is possible for a partner to select a behaviour by sending on r the corresponding label ($r \triangleleft l.P$). The try-catch construct $\mathsf{try}(\tilde{r})\{P\}\ \mathsf{catch}\ \{Q\}$ describes a process P (called *default process*) that communicates on the private channels $\tilde{r}$. If some exception is thrown on $\tilde{r}$ before P has ended, the compensation handler Q is going to take over. The construct $\mathsf{throw}(\tilde{r})$ throws the exception on the channels $\tilde{r}$. Try-catch blocks can be nested, but internal blocks must involve a proper subset of the set of argument channels of the outer try block (this is enforced by the type system). The idea behind this condition is that internal try blocks involve smaller sets of participants using a subset of channels. The exception raised inside these blocks can be resolved by the involved peers without affecting the whole system (or in general a wider sets of participants). If something goes wrong during the execution of the handler, that is the exception cannot be solved “internally” anymore, the control can be passed to the handler of the outer try block by throwing an exception on r (see example 2).

As in [9], in order to model TCP-like asynchronous communications (with non-blocking send but message order preservation between a given pair of participants), we use *queues of messages* L. Queues have a bidimensional structure, this is necessary to guarantee communication consistency: when performing an action on a channel s^{φ}, a process is going to write or read at the level φ of the queue associated to s. However in the user written code the level is always zero ($\varphi = 0$), namely the

P,Q	::=	$\overline{a}[2..n](\tilde{s}).P$	Multicast Request	|	if e then P else P	Conditional
	|	$a[\mathtt{p}](\tilde{s}).P$	Accept	|	$P \mid P$	Parallel
	|	$r!\langle\tilde{e}\rangle$	Output	|	$P; P$	Sequencing
	|	$r?(\tilde{x}).P$	Input	|	$\mathbf{0}$	Inaction
	|	$r \triangleleft l.P$	Select	|	$(\nu n)P$	Hiding
	|	$r \triangleright \{l_i : P_i\}_{i\in I}$	Branch	|	def D in P	Recursion
	|	try$(\tilde{r})\{P\}$ catch $\{P\}$	Try-Catch	|	$X\langle\tilde{e}\tilde{s}\rangle$	Process call
	|	throw$(\tilde{r})$	Throw	|	$s : L$	Named queue

v	::=	a | true | false	Value	D	::=	$\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$	Declaration
e	::=	v | x| e and e' | not e ...	Expression	L	::=	$l \cdot L \mid \tilde{v} \cdot L \mid \varnothing$	Queue

Figure 1 Syntax

programmer is not responsible of managing the queue levels, which are increased automatically at runtime during the evaluation (for more detail see Section 3). For a consistent semantics we require that a service can never occur in a try-catch block (this is enforced by the type system), otherwise if an exception is captured and the handler is executed, the session inside the default process will disappear while having still some pending communications. We believe this is only an apparent limitation, because all the practical examples we encountered can be easily implemented in our language.

► **Example 1.** We present here a three-party use case that models criss-crossing global interaction [5], which can be coded as in Figure 2 where we use different fonts for *variables* and `values`.

The Seller receives an order from a Client, then, inside a try block, he processes the order and then sends back the confirmation. The Client waits for the order confirmation until, at some point, he decides he has waited too long by throwing an exception. The handler of the Seller checks if the confirmation has been sent by the Seller: if it has not (i.e., *conf* = false) then the interaction is aborted by the Seller; otherwise *conf* = true means that the Client has raised the exception before receiving the confirmation from the Seller; in this case execution goes on with $P_{\mathtt{OK}}$ and $P'_{\mathtt{OK}}$ respectively. Thus *conf* = true corresponds to order completion: from this moment on the interaction must proceed even if the Client has decided differently (this is quite standard in business protocol specifications: when a client aborts too late the transaction he is often compelled either to conclude the payment or to pay some penalty fees). The above interaction contains an escape for the Client who has the right to abort the transaction if the Seller is late for delivery. In $P'_{\mathtt{OK}}$ the Client sends the code of his Bank account to the Bank. The Bank checks if there is enough money then, according to the result of the test, sends `OK` or `NEM` (`Not Enough Money`) to the Client. If the answer is `OK`, the Bank sends `OK` also to the Seller who sends the delivery date to the Client. If the answer is `NEM`, the Client and the Bank start to deal for a loan. Now let us concentrate on the Client-Bank deal. The Bank refreshes the offer every n-seconds, where n is the value of *timeout*. The process iterates until an agreement is reached. The time intervals are modelled through a *timer* construct (let h = `timer`(0) in ...). Thus there are two iterations in the process: one related to the deal (def X) and the other used by the Bank to iterate on time intervals (def Y). The Client examines the Bank offer and, if he agrees on it, he throws an exception to exit the iteration (this is implemented as a inner try-catch block involving only s_2: when the throw is raised the Seller is not involved) otherwise he waits for another offer and iterates the negotiation. Dually the Bank sets the timer to 0 at each deal iteration; the internal recursion iterates until the time-out is reached and then the loan offer is updated. The Bank calculates another offer then iterates the outer recursive process sending the new loan to the Client. An interesting scenario is when the time-out and the acceptance of the loan from the Client rise concurrently. It can then happen that the Client has accepted an offer while the Bank was updating his offer after a time-out: the Bank and the Client agreed on different amounts of money. This is resolved by the handlers P_1

$$
\begin{array}{rcl}
\mathtt{Seller} & = & \mathtt{BS}[1](s_1, s_2).s_1?(order).\mathsf{try}(s_1, s_2)\{\text{ elaborate order }; conf = \mathtt{true}; s_1!\langle \mathtt{true} \rangle; P_{\mathtt{OK}}\} \\
& & \qquad\qquad \mathsf{catch}\ \{\mathsf{try}(s_1, s_2)\{\mathsf{if}\ conf\ \mathsf{then}\ P_{\mathtt{OK}}\ \mathsf{else}\ \ \mathsf{throw}(s_1, s_2)\}\ \mathsf{catch}\ \{\mathtt{abort}\}\} \\
P_{\mathtt{OK}} & = & s_1 \rhd \{\mathtt{OK} : s_1!\langle \mathtt{date} \rangle, \mathtt{NOK} : \mathsf{throw}(s_1, s_2)\} \\
\mathtt{Client} & = & \overline{\mathtt{BS}}[2](s_1, s_2).s_1!\langle \mathtt{order} \rangle; \mathsf{try}(s_1, s_2)\{s_1?(conf).P'_{\mathtt{OK}} \mid \mathsf{throw}(s_1, s_2))\} \\
& & \qquad\qquad \mathsf{catch}\ \{\mathsf{try}(s_1, s_2)\{P'_{\mathtt{OK}}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\} \\
P'_{\mathtt{OK}} & = & s_1!\langle \mathtt{code} \rangle; s_1 \rhd \{\mathtt{OK} : s_1?(date).\mathbf{0}, \mathtt{NEM} : s_1!\langle \mathtt{rf} \rangle; s_1?(f). \\
& & \mathsf{try}(s_2)\{\mathsf{def} X(s_2) = \mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ \mathsf{throw}(s_2)\ \mathsf{else}\ \ s_2?(f).X < s_2 > \mathsf{in} X < s_2 > \mathsf{catch}\{P_1\}\} \\
P_1 & = & s_2!\langle \mathtt{f} \rangle; s_1 \rhd \{\mathtt{OK} : s_2?(date), \mathtt{NOK} : \mathbf{0})\} \\
\mathtt{Bank} & = & \overline{\mathtt{BS}}[3](s_1, s_2).\mathsf{try}(s_2)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathsf{try}(s_1, s_2)\{P''_{\mathtt{OK}}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\} \\
P''_{\mathtt{OK}} & = & s_1?(code).\ \mathsf{if}\ enoughmoney\ \mathsf{then}\ s_1 \lhd \mathtt{OK}.s_1 \lhd \mathtt{OK}. \\
& & \ \mathsf{else}\ s_1 \lhd \mathtt{NEM}.s_1?(rf).s_1!\langle \mathtt{f} \rangle; \mathsf{try}(s_2)\{\mathsf{def} X(s_2) =\ \ \mathsf{let}\ h = \mathtt{timer}(0)\ \ \mathsf{in}\ \mathsf{def} Y(s_2) = \\
& & \mathsf{if}\ h = timeout\ \mathsf{then}\ \mathsf{calculate}\{\mathtt{f}\}; s_2!\langle \mathtt{f} \rangle; X < s_2 >\ \ \mathsf{else} \\
& & Y < s_2 >\ \ \mathsf{in} Y < s_2 > \mathsf{in} X < s_2 >\}\ \mathsf{catch}\ \{P_2\} \\
P_2 & = & s_2?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ s_1 \lhd \mathtt{OK}.s_1 \lhd \mathtt{OK}.\ \mathsf{else}\ \ s_1 \lhd \mathtt{NOK}.s_1 \lhd \mathtt{NOK}.
\end{array}
$$

Figure 2 Client-Seller-Bank code

and P_2: after the exception has been thrown the Bank sends to the Client the latest value of the loan. The Client checks it and decide whether to accept it or not. In the latter case, being out of money, he sends a NOK label to the Seller who aborts the transaction by throwing an exception.

3 Operational Semantics for Multiparty Exceptions

We extend the semantics of multiparty sessions with exception handling. Exceptions can be raised inside try-catch blocks by means of the throw construct. We ensure that conversations are properly carried on by increasing the level of involved channels in case of exception: handlers communicate on a level $\varphi+1$ while pending messages (i.e. that are sent by main processes before passing the execution to the handlers) are sent via channels of level φ. Reduction rules are defined in Figure 3. The reduction system uses an *Exception Environment* Σ, which keeps track of the raised exceptions. This will be used in rules [*Thr*], [*RThr*],[*ZThr*].

Reduction rules use evaluation contexts defined by the following grammars:

$$C := [\ ]\ \mid\ \mathsf{def}\ D\ \mathsf{in}\ C\ \mid\ C; P \qquad \mathcal{E} := C\ \mid\ \mathcal{E} \mid P\ \mid\ (\nu n)\mathcal{E}\ \mid\ \mathsf{try}(\tilde{r})\{\mathcal{E}\}\ \mathsf{catch}\ \{Q\}$$

with the usual semantics: if a process P reduces to P', then $\mathcal{E}[P]$ reduces to $\mathcal{E}[P']$.

Rule [*Link*] establishes the connection among n peers on the private channels $\tilde{s}$, to be used for the communications within the session. One queue for each private channel is produced.

Rules [*Send*], [*Sel*], [*Recv*], [*Branch*] are defined as usual, except for the fact that they put and get values/labels from the φth level of the queue. At first the processes read and write at the first level of the queue ($\varphi = 0$). When a process catches an exception, the indexes of all the occurrences of channels involved are increased by one and the exception is propagated to the other peers, which can safely continue to modify the queues at the previous level until they receive the exception. Those messages delivered to the out-of-date level of the queues will be ignored by the peers that have already caught the exception and increased their queue levels.

Rule [*Thr*] applies when the exception is thrown locally. In this rule a $\mathsf{throw}(\tilde{r})$ is added to the environment in order to acknowledge all the try-catch blocks on the same set of channels $\tilde{r}$. This environment update is performed unless some other $\mathsf{throw}(\tilde{r}')$, with $\tilde{r} \subseteq \tilde{r}'$ is already in Σ, because this would mean that an exception may be caught by an embedding block, causing the current try-catch block to disappear. In that case throwing the exception on channels $\tilde{r}$ would be useless and possibly

$$\Sigma \vdash C_1[\overline{a}[2..n](\tilde{s}).P_1] \mid C_2[a[2](\tilde{s}).P_2] \mid ... \mid C_n[a[n](\tilde{s}).P_n] \longrightarrow \Sigma \vdash (\nu\tilde{s})\,(C_1[P_1] \mid C_2[P_2] \mid ... \mid C_n[P_n] \mid s_1 : \varnothing \mid ... \mid s_m : \varnothing) \qquad [Link]$$

$$\Sigma \vdash \mathcal{E}[s^{\varphi}!\langle\tilde{e}\rangle] \mid s[\varphi] : L \longrightarrow \Sigma \vdash \mathcal{E} \mid s[\varphi] : (L :: \tilde{v}) \quad (\tilde{e} \downarrow \tilde{v}) \qquad [Send]$$

$$\Sigma \vdash \mathcal{E}[s^{\varphi} \lhd l.P] \mid s[\varphi] : L \longrightarrow \Sigma \vdash \mathcal{E}[P] \mid s[\varphi] : (L :: l) \qquad [Sel]$$

$$\Sigma \vdash \mathcal{E}[s^{\varphi}?(\tilde{x}).P] \mid s[\varphi] : (\tilde{v} :: L) \longrightarrow \Sigma \vdash \mathcal{E}[P\{\tilde{v}/\tilde{x}\}] \mid s[\varphi] : L \qquad [Recv]$$

$$\Sigma \vdash \mathcal{E}[s^{\varphi} \rhd \{l_i : P_i\}_{i\in I}] \mid s[\varphi] : (l_{i_0} :: L) \longrightarrow \Sigma \vdash \mathcal{E}[P_{i_0}] \mid s[\varphi] : L \quad (i_0 \in I) \qquad [Branch]$$

$$\Sigma \vdash \mathsf{try}(\tilde{r})\{C[\mathsf{throw}(\tilde{r})] \mid P\}\ \mathsf{catch}\ \{Q\} \longrightarrow \Sigma \uplus \mathsf{throw}(\tilde{r}) \vdash \mathsf{try}(\tilde{r})\{C \mid P\}\ \mathsf{catch}\ \{Q\} \qquad [Thr]$$

$$\Sigma, \mathsf{throw}(\tilde{r}) \vdash \mathsf{try}(\tilde{r})\{P\}\ \mathsf{catch}\ \{Q\} \longrightarrow \Sigma, \mathsf{throw}(\tilde{r}) \vdash Q\{s^{\varphi+1}/s^{\varphi}\}_{s^{\varphi}\in\tilde{r}} \quad (\mathsf{throw}(\tilde{r}') \in \Sigma\ \mathit{implies}\ \mathsf{try}(\tilde{r}')\ldots \notin P,\ \tilde{r}' \subseteq \tilde{r}) \qquad [RThr]$$

$$\Sigma \vdash (\nu\tilde{s})(\textstyle\prod_i \mathcal{E}_i[\mathsf{try}(\tilde{r})\{\mathbf{0}\}\ \mathsf{catch}\ \{Q_i\}])_{i\in 1..n} \longrightarrow \Sigma \vdash (\nu\tilde{s})(\prod_i \mathcal{E}_i)_{i\in 1..n} \quad (\mathsf{throw}(\tilde{r}) \notin \Sigma) \qquad [ZThr]$$

Figure 3 Reduction rules

dangerous (leading to some inconsistency on the queue levels).

The operation of environment update is defined as follows:

$$\Sigma \uplus \mathsf{throw}(\tilde{r}) = \begin{cases} \Sigma & \text{if } \mathsf{throw}(\tilde{r}') \in \Sigma, \natural(\tilde{r}) \subseteq \tilde{r}' \\ \Sigma \cup \mathsf{throw}(\natural(\tilde{r})) & \text{otherwise.} \end{cases}$$

where, $\mathsf{throw}(\tilde{r})$ is added to the environment only if there are no other throw on a bigger or equal set of channels. We use $\natural(\tilde{r})$ to level down the indexes of the channels in $\tilde{r}$, as we can see in the following definition: $\natural(s_1^{\varphi_1}, \ldots, s_n^{\varphi_n}) = s_1^{\varphi}, \ldots, s_n^{\varphi}$, where $\varphi = min(\varphi_1, \ldots, \varphi_n)$.
The need of this operation is made clear by Example 2.

Rule [*RThr*] applies when an exception has been thrown and therefore it can be found in Σ. In this case, the try block reduces to the handler, where all the queue levels are updated. Now let us explain the side condition. The try block reduces only if no inner exception can be caught (i.e., if $\mathsf{throw}(\tilde{r}') \in \Sigma$): this would mean that another peer could be executing the internal handler. In order to be consistent with it, the current process must catch the same internal exception before catching the external one. Then, when all the internal exceptions have been caught, even if a $\mathsf{throw}(\tilde{r}')$ does occur in Σ, it can be ignored, and the handler corresponding to the exception on $\tilde{r}$ can take over.

Rule [*ZThr*] deals with the cases in which the default process in a try block has been reduced to $\mathbf{0}$ and no pending exception occurs in Σ. No such completed try-catch block can be reduced to the inaction, until every other peer has completed the corresponding try-block and are ready to continue the execution. The reason is that even if one try-block has terminated, one among its communicating peers could throw an exception and then the handlers have to interact. So we consider the try-catch blocks in each peer at the same level, when every peer has terminated then they all can go on. Since we consider only well-typed processes, and every process is type-checked with respect to the same global type, it is safe to assume that if we have n communicating peers, then there will be n try-catch blocks to be synchronized.

Rules for conditionals and recursive definitions are standard. Moreover, as usual, we consider processes modulo structural congruence. Besides the standard structural rules, we define the following one: $\mathsf{try}(\tilde{r})\{(\nu n)P\}\ \mathsf{catch}\ \{Q\} \equiv (\nu n)\mathsf{try}(\tilde{r})\{P\}\ \mathsf{catch}\ \{Q\}$ if $n \notin \mathrm{fn}(Q)$.

The complete set of rules can be found in [3].

► **Example 2.** Let us consider the reduction of the following process:

$$\emptyset \vdash \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{\mathsf{throw}(s_1) \mid P\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1, s_2)\}\}\ \mathsf{catch}\ \{Q\} \mid \mathsf{try}(s_1, s_2)\{\mathsf{throw}(s_1, s_2) \mid \mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{Q'\}\}\ \mathsf{catch}\ \{Q''\}$$

In the first line, in the inner try-catch block both the default process and the handler contain a throw and the latter' is on (s_1, s_2). In this way we can model a situation in which if the handling of the enclosed exception fails, the outer block is alerted to handle the failure. The default process in the second line contains a throw on (s_1, s_2).
Case 1 Let us suppose that $\mathsf{throw}(s_1)$ is raised first, by applying rule $[Thr]$:

$$\begin{array}{lll}\mathsf{throw}(s_1) & \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1,s_2) \mid P'\}\}\ \mathsf{catch}\ \{Q\} \mid \\ & & \mathsf{try}(s_1,s_2)\{\mathsf{throw}(s_1,s_2) \mid \mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{Q'\}\}\ \mathsf{catch}\ \{Q''\}\end{array}$$

Then $\mathsf{throw}(s_1, s_2)$ is raised and we apply $[Thr]$ again:

$$\begin{array}{lll}\mathsf{throw}(s_1), \mathsf{throw}(s_1,s_2) & \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1,s_2) \mid P'\}\}\ \mathsf{catch}\ \{Q\} \mid \\ & & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{Q'\}\}\ \mathsf{catch}\ \{Q''\}\end{array}$$

Then, by rule $[RThr]$, the only exception that can be thrown is the one corresponding to $\mathsf{throw}(s_1)$, since, because of the side condition, the external try-blocks on (s_1, s_2) cannot reduce:

$$\begin{array}{lll}\mathsf{throw}(s_1), \mathsf{throw}(s_1,s_2) & \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{throw}(s_1^1,s_2) \mid P'\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q\} \mid \\ & & \mathsf{try}(s_1,s_2)\{Q'\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q''\}\end{array}$$

Notice that, because of the queue level updating, the throw that was part of the inner handler is now $\mathsf{throw}(s_1^1, s_2)$. If something goes wrong in handling the inner exception, $\mathsf{throw}(s_1^1, s_2)$ must reduce even if the level of the channel arguments and of the outer try block do not match: this example shows that the mismatch is due to the fact that some of the channels were involved in a failed exception handling. To balance the levels of the channels in the throw we use the operation $\natural$ which flats all levels to the minimum.
Now we apply rule $[Thr]$ again. Since $\mathsf{throw}(\natural(s_1^1, s_2)) = \mathsf{throw}(s_1, s_2)$, the environment Σ does not change:

$$\mathsf{throw}(s_1), \mathsf{throw}(s_1,s_2) \vdash \mathsf{try}(s_1,s_2)\{P'\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q\} \mid \mathsf{try}(s_1,s_2)\{Q'\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q''\}$$

Finally, by rule $[RThr]$, the try blocks on s_1, s_2 can reduce:

$$\mathsf{throw}(s_1), \mathsf{throw}(s_1,s_2) \vdash Q\{s_1^2, s_2^1/s_1^1, s_2\} \mid Q''\{s_1^2, s_2^1/s_1^1, s_2\}$$

Notice that the channels in Q and Q'' are all at the same level.

Case 2 Now let us suppose that $\mathsf{throw}(s_1, s_2)$ is raised first:

$$\begin{array}{lll}\mathsf{throw}(s_1,s_2) & \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1,s_2) \mid P'\}\}\ \mathsf{catch}\ \{Q\} \mid \\ & & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{Q'\}\}\ \mathsf{catch}\ \{Q''\}\end{array}$$

We apply rule $[Thr]$ again to the inner $\mathsf{throw}(s_1)$. Notice that $\mathsf{throw}(s_1)$ is not added to the environment since an enclosing throw is already present.

$$\begin{array}{lll}\mathsf{throw}(s_1,s_2) & \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1,s_2) \mid P'\}\}\ \mathsf{catch}\ \{Q\} \mid \\ & & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P\}\ \mathsf{catch}\ \{Q'\}\}\ \mathsf{catch}\ \{Q''\}\end{array}$$

then rule $[RThr]$ is applied twice:

$$\mathsf{throw}(s_1,s_2) \vdash Q\{s_1^1, s_2^1/s_1, s_2\} \mid Q''\{s_1^1, s_2^1/s_1, s_2\}$$

.

4 Typing Structured Global Escapes

This section extends the type system in [9] to exception handling constructs. In particular the goals of the system are:

(i) to check that the enclosed try-catch block is listening on a smaller set of channels. We want to enforce this condition to enable the independence of the components w.r.t. exceptions: if an exception is captured by an inner component, this is not going to affect the enclosing ones.

(ii) to check that no session request or accept occurs inside a try-catch block as we explained in Section 2.
(iii) to check that throws were written by the programmer in the right position, that is a $\mathsf{throw}(\tilde{r})$ must be at the top level of the default process in a try block on the channels $\tilde{r}$.
For lack of space we cannot be self contained w.r.t. the original system so we just describe the new features. The full type system can be found in [3].

Types. In defining the syntax of our types, we distinguish between *global types*, ranged over by G, which describe the whole communication of a multiparty session, and *end-point types*, ranged over by A, which describe the communication from the point of view of a single participant.

The grammar of a global type is as follows:

Partial	γ	$::=$	$\mathtt{p}_1 \to \mathtt{p}_2 : k\langle\tilde{S}\rangle \mid \mathtt{p}_1 \to \mathtt{p}_2 : k\{l_i : \gamma_i\}_{i\in I} \mid [\tilde{k}, \gamma, \gamma] \mid \gamma; \gamma \mid \gamma \parallel \gamma \mid \mu\mathbf{t}.\gamma \mid \mathbf{t} \mid \epsilon$
Global	G	$::=$	$\gamma; \mathsf{end}$
Sorts	S	$::=$	$\mathsf{bool} \mid \ldots \mid \langle G\rangle$

A global type G is an ended partial type γ. The type $\mathtt{p}_1 \to \mathtt{p}_2 : k\langle\tilde{S}\rangle$ says that participant $\mathtt{p}_1$ sends values of sort $\tilde{S}$ to participant $\mathtt{p}_2$ over the channel k (represented as a natural number). The type $\mathtt{p}_1 \to \mathtt{p}_2 : k\{l_i : \gamma_i\}_{i\in I}$ says that participant $\mathtt{p}_1$ sends one of the labels l_i to participant $\mathtt{p}_2$ over the channel k. If the label l_j is sent, the conversation continues as the corresponding γ_j describes. The type $[\tilde{k}, \gamma, \gamma']$ says that the conversation specified by γ is performed, unless some exception involving channels $\tilde{k}$ arises. In this case the conversation γ' takes over. Moreover, types can be composed by sequential and parallel composition, and they can be recursively defined.

The grammar of an end-point type is as follows:

Partial action	α,β	$::=$	$k!(\tilde{S}) \mid k?(\tilde{S})$	*send and receive*
		$\mid$	$k\oplus\{l_i : \alpha_i\}_{i\in I} \mid k\&\{l_i : \alpha_i\}_{i\in I}$	*selection and branching*
		$\mid$	$[\tilde{k}, \alpha, \alpha]$	*try-catch*
		$\mid$	$\mu\mathbf{t}.\alpha \mid \mathbf{t}$	*recursion and type variable*
		$\mid$	$\epsilon \mid \alpha; \alpha$	*inaction and sequencing*
Action	A, B	$::=$	$\alpha \mid \alpha; \mathsf{end} \mid \mathsf{end}$	

As in [9], a session type records the identity number of the session channel it uses at each action type, and we use the *located* type $A@\mathtt{p}$ to represent the end-point type A assigned to participant $\mathtt{p}$.
Types $k!(\tilde{S})$ and $k?(\tilde{S})$ represent output and input of values of type $\tilde{S}$ at s_k. Types $k \oplus \{l_i : \alpha_i\}_{i\in I}$ and $k\&\{l_i : \alpha_i\}_{i\in I}$ describe selection and branching: the former selects one of the labels provided by the latter, say l_i at k then they behave as α_i . The remaining types are a local version of the global ones.

Projection. As usual we define a projection that given a global type G and a participant $\mathtt{p}$ returns the end-point corresponding to the local behaviour of $\mathtt{p}$. We write $G \upharpoonright \mathtt{p}$ to denote such a projection.

The projection is defined first over global types and then over partial global types, that is $(\gamma; \mathsf{end}) \upharpoonright \mathtt{p} = (\gamma \upharpoonright \mathtt{p}).\mathsf{end}$.

The exception type is projected in every participant as a local exception type, even if the participant has no activity in the try-catch block (we call these inactive blocks *dummy try-catch*). This to guarantee that the structure of try-catch blocks is the same in each process.

$$([\tilde{k}, \gamma_1, \gamma_2]) \upharpoonright \mathtt{p} = [\tilde{k}, (\gamma_1 \upharpoonright \mathtt{p}), \gamma_2 \upharpoonright \mathtt{p}] \;\text{ if } \mathsf{ch}(\gamma_i) \subseteq k \text{ and } \left([\tilde{k}', \gamma_1', \gamma_2'] \in \gamma_i \text{ implies } \tilde{k}' \subset \tilde{k}\right)$$

When the side condition does not hold the map is undefined.

Typing Rules. Type assumptions over names and variables are stored into the *Standard environment* Γ that stores type assumptions over names and variables, and is defined by $\Gamma ::= \emptyset \mid \Gamma, u : S \mid \Gamma, X : \tilde{S}\tilde{A}$, and into the *Session environment* Δ that records session types associated to session channels and is defined by $\Delta ::= \emptyset \mid \Delta, \tilde{k} : \{A@\mathtt{p}\}_{\mathtt{p}\in I}$.

$$\frac{}{\Gamma, a : S \vdash a : S}\lfloor\textsc{Name}\rfloor \qquad \frac{\Delta \text{ end only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta\,,\, \tilde{z}}\lfloor\textsc{Inact}\rfloor \qquad \frac{\Gamma, a : \langle G\rangle \vdash P \triangleright \Delta\,,\, \tilde{z}}{\Gamma \vdash (\nu a)P \triangleright \Delta\,,\, \tilde{z}}\lfloor\textsc{NRes}\rfloor$$

$$\frac{\Gamma \vdash a : \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta \cup \{\tilde{s} : (G \upharpoonright 1)@1\}\,,\, - \qquad |\tilde{s}| = |\mathsf{sid}(G)|}{\Gamma \vdash \overline{a}[2..n](\tilde{s}).P \triangleright \Delta\,,\, -}\lfloor\textsc{MReq}\rfloor$$

$$\frac{\Gamma \vdash a : \langle G\rangle \quad \Gamma \vdash P \triangleright \Delta \cup \{\tilde{s} : (G \upharpoonright \mathrm{p})@\mathrm{p}\}\,,\, - \qquad |\tilde{s}| = |\mathsf{sid}(G)|}{\Gamma \vdash a[\mathrm{p}](\tilde{s}).P \triangleright \Delta\,,\, -}\lfloor\textsc{MAcc}\rfloor$$

$$\frac{\forall j. \Gamma \vdash e_j : S_j}{\Gamma \vdash s_k^{\varphi}!\langle\tilde{e}\rangle \triangleright \{\tilde{s} : k^{\varphi}!(\tilde{S})@\mathrm{p}\}\,,\, \tilde{z}}\lfloor\textsc{Send}\rfloor \qquad \frac{\Gamma, \tilde{x} : \tilde{S} \vdash P \triangleright \Delta \cup \{\tilde{s} : A@\mathrm{p}\}\,,\, \tilde{z}}{\Gamma \vdash s_k^{\varphi}?(\tilde{x}).P \triangleright \Delta \cup \{\tilde{s} : k^{\varphi}?(\tilde{S}); A@\mathrm{p}\}\,,\, \tilde{z}}\lfloor\textsc{Rcv}\rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta \cup \{\tilde{s} : A_j@\mathrm{p}\}\,,\, \tilde{z} \qquad j \in I}{\Gamma \vdash s_k^{\varphi} \lhd l_j.P \triangleright \Delta \cup \{\tilde{s} : k^{\varphi} \oplus \{l_i : \alpha_i\}_{i\in I}@\mathrm{p}\}\,,\, \tilde{z}}\lfloor\textsc{Sel}\rfloor \qquad \frac{\Gamma \vdash P_i \triangleright \Delta \cup \{\tilde{s} : A_i@\mathrm{p}\}\,,\, \tilde{z} \qquad \forall i \in I}{\Gamma \vdash s_k^{\varphi} \rhd \{l_i : P_i\}_{i\in I} \triangleright \Delta \cup \{\tilde{s} : k^{\varphi}\&\{l_i : \alpha_i\}_{i\in I}@\mathrm{p}\}\,,\, \tilde{z}}\lfloor\textsc{Branch}\rfloor$$

$$\frac{\begin{array}{c}\Gamma \vdash P \triangleright \{\tilde{s} : \alpha@\mathrm{p}\}\,,\, \tilde{z} \quad \Gamma \vdash Q \triangleright \{\tilde{s} : \beta@\mathrm{p}\}\,,\, \tilde{z}' \\ \mathsf{ch}(P) \subseteq \tilde{z} \quad (\tilde{z}' \neq -) \Rightarrow \tilde{z} \subset \tilde{z}'\end{array}}{\Gamma \vdash \mathsf{try}(\tilde{z})\{P\} \text{ catch } \{Q\} \triangleright \{\tilde{s} : [\tilde{z}, \alpha, \beta]@\mathrm{p}\}\,,\, \tilde{z}'}\lfloor\textsc{Try}\rfloor \qquad \frac{\natural(\tilde{r}) = \natural(\tilde{z})}{\Gamma \vdash \mathsf{throw}(\tilde{r}) \triangleright \emptyset\,,\, \tilde{z}}\lfloor\textsc{Throw}\rfloor$$

$$\frac{\Gamma \vdash P \triangleright \Delta\,,\, \tilde{z} \quad \Gamma \vdash Q \triangleright \Delta'\,,\, \tilde{z} \quad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'\,,\, \tilde{z}}\lfloor\textsc{Par}\rfloor \qquad \frac{\Gamma \vdash P \triangleright \Delta\,,\, \tilde{z} \quad \Gamma \vdash Q \triangleright \Delta'\,,\, \tilde{z}}{\Gamma \vdash P; Q \triangleright \Delta \cdot \Delta'\,,\, \tilde{z}}\lfloor\textsc{Seq}\rfloor$$

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta\,,\, \tilde{z} \quad \Gamma \vdash Q \triangleright \Delta\,,\, \tilde{z}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta\,,\, \tilde{z}}\lfloor\textsc{If}\rfloor$$

$$\frac{\Gamma \vdash \tilde{e} : \tilde{S} \qquad \Delta \text{ end only}}{\Gamma, X : \tilde{S}\tilde{A} \vdash X\langle\tilde{e}\tilde{s}_1..\tilde{s}_n\rangle \triangleright \Delta, \tilde{s}_1 : A_1@\mathrm{p}_1, .., \tilde{s}_n : A_n@\mathrm{p}_n\,,\, \tilde{z}}\lfloor\textsc{Var}\rfloor$$

$$\frac{\Gamma, X : \tilde{S}\tilde{A}, \tilde{x} : \tilde{S} \vdash P \triangleright \tilde{s}_1 : A_1@\mathrm{p}_1, .., \tilde{s}_n : A_n@\mathrm{p}_n\,,\, \tilde{z} \qquad \Gamma, X : \tilde{S}\tilde{A} \vdash Q \triangleright \Delta\,,\, \tilde{z}}{\Gamma \vdash \mathsf{def}\ X(\tilde{x}\tilde{s}_1..\tilde{s}_n) = P \text{ in } Q \triangleright \Delta\,,\, \tilde{z}}\lfloor\textsc{Def}\rfloor$$

Figure 4 Typing rules

The typing judgement has the form: $\Gamma \vdash P \triangleright \Delta\,,\, \tilde{z}$, where P is the process to be typed and $\tilde{z}$ refers to the channels on which the enclosing try-catch block is listening for an exception. We need to take track of those channels in order to ensure (i), (ii) and (iii).

The complete set of typing rules is given in Figure 4. All the rules are standard w.r.t. multiparty sessions theory, except for the two rules ⌊Try⌋ and ⌊Throw⌋.

In rule ⌊Try⌋ the default process P and the exception handler Q are both typed with a session environment composed by $\tilde{s}$ channels only, this is to guarantee that no other communications on channels belonging to some other sessions would be interrupted due to the raising of an exception. The channels $\tilde{z}'$ refers to the channels on which an eventual external try in listening for exceptions. Notice that $\tilde{z}'$ may be an empty sequence, that is the try-block that is being type-checked is at the top level. The default process P considers in its typing the sequence $\tilde{z}$ of the current try-block, while

the exception handler consider the sequence $\tilde{z}'$ of the external try-catch block: this is because when executing the exception handler would be directly enclosed in the external try-catch block. Notice that if $\tilde{z}'$ is different from the empty sequence ($\tilde{z}' \neq -$) then $\tilde{z}$ must be a strict subsequence, this is to guarantee that the current try-catch block is listening on a smaller set of channels w.r.t. the enclosing one. The rule also checks that the channels on which P is communicating ($\mathsf{ch}(P)$) are included in $\tilde{z}$: this is to ensure that all the channels involved in some communication in P will be notified of the exception.

In rule $\lfloor$THROW$\rfloor$ we check that a $\mathsf{throw}(\tilde{r})$ has been written at the right position: it must be at the top level of a default process in a try-block on channels $\tilde{r}$. That is the $\tilde{z}$ recorded in the judgement must be the same as $\tilde{r}$ (up to the $\natural$ operation). Note that given closed annotated processes (i.e. bound variables and names are annotated by types), the type checking is decidable (since checking coherent of G is polymonial with respect to the size of G [7]).

Properties. The type discipline ensures, as in previous session types literature [8, 9]:

- the lack of standard type errors in expressions (***Subject Reduction***):
- communication error freedom (***Communication Safety***),
- the interactions of a typable process exactly follow the specification described by its global type (***Session Fidelity***), and
- once a communication has been established, well-typed programs will never stuck at communication points (***Progress***)

► Theorem 4.1 (Subject Congruence and Reduction). Suppose $\Gamma \vdash P \triangleright \emptyset$. Then *(1)* $P \equiv P'$ implies $\Gamma \vdash P' \triangleright \emptyset$; and *(2)* $P \to P'$ implies $\Gamma \vdash P' \triangleright \emptyset$.

For the proof, see [3]. From this theorem, ***Session Fidelity Theorem*** (the interaction of a typable process exactly follow the specification described by its global type) is straightforward. We also state the two theorems which are derived following the technique developed in [9]; see [3] for the proofs.

Below we write $P\langle r!\rangle$ (resp. $P\langle r?\rangle$) if P contains an emitting (resp. receiving) active prefix at r up to $\equiv$, and we say that P has a *redex at r* if it has an active prefix at r among its redexes (Section 5 in [9] gives further details for the redexes). The reduction context $\mathcal{E}$ is defined in Section 3.

► Theorem 4.2 (Communication Safety). Suppose $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$ s.t. Δ is coherent and P has a redex at free s^{φ}. Then:

1. (linearity) $P \equiv \mathcal{E}[s[\varphi] : \tilde{h}]$ such that either
   a. $P\langle s^{\varphi}?\rangle$, s^{φ} occurs exactly once in $\mathcal{E}$ and $\tilde{h} \neq \emptyset$; or
   b. $P\langle s^{\varphi}!\rangle$ and s^{φ} occurs exactly once in $\mathcal{E}$; or
   c. $P\langle s^{\varphi}?\rangle$, $P\langle s^{\varphi}!\rangle$, and s^{φ} occurs exactly twice in $\mathcal{E}$.
2. (error-freedom) if $P \equiv \mathcal{E}[R]$ with $R\langle s^{\varphi}?\rangle$ being a redex:
   a. If $R \equiv s^{\varphi}?(\tilde{y}); Q$ then $P \equiv \mathcal{E}'[s[\varphi] : \tilde{v} \cdot \tilde{h}]$ for some $\mathcal{E}'$ and $|\tilde{v}| = |\tilde{y}|$.
   b. If $R \equiv s^{\varphi} \triangleright \{l_i : Q_i\}_{i \in I}$ then $P \equiv \mathcal{E}'[s[\varphi] : l_j \cdot \tilde{h}]$ for some $\mathcal{E}'$ and $j \in I$.

The type discipline ensures also the progress property. A process is simple when each prefixed subterm in it has only a unique session.

► Definition 4.3 (simple). A process P is *simple* when it is typable with a type derivation where the session typing in the premise and the conclusion of each prefix rule is restricted to at most a singleton.

In a simple well-linked P, each session is never hindered by other sessions nor by a name prefixing:

► Definition 4.4 (well-linked). We say P is *well-linked* when for each $P \to^{*} Q$, whenever Q has an active prefix whose subject is a (free or bound) shared name, then it is always part of a redex.

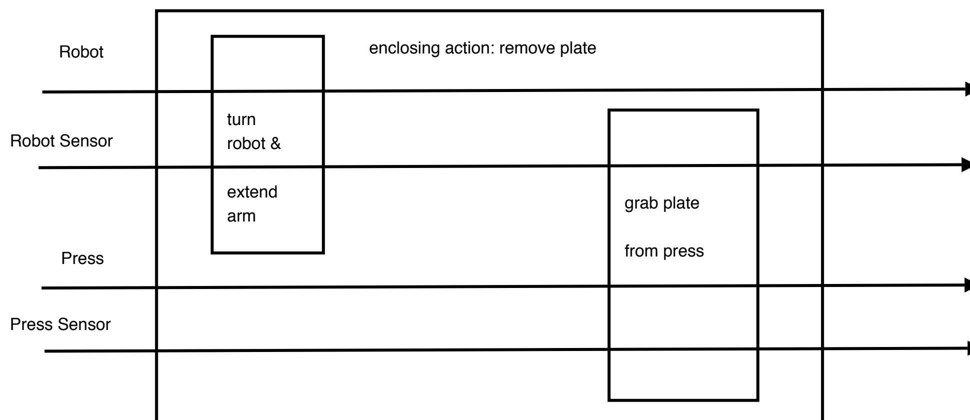

Figure 5 The coordinated action `remove-plate`.

Then the following theorem states that a progress holds for a simple process with a queue for each session channel, such that each prefixed subterm in it has only a unique session and such that each session is never hindered by other sessions nor by a name prefixing.

▶ Theorem 4.5 (Progress). Let P be a simple and well-linked program and $\Gamma \vdash P \triangleright \emptyset$. Then P has the *progress property* in the sense that $P \longrightarrow^{*} P'$ implies either $P' \equiv \mathbf{0}$ or $P' \longrightarrow P''$ for some P''.

5 Coordinated Exception Handling and Resolution: an Example

Coordinated Atomic Actions. In the context of distributed object systems, *Coordinated Atomic Actions* (CA*s*) [13] represent conversation units w.r.t. resource access and recovery activities, in the sense that when an exception is thrown inside a CA the handling is confined within the CA participants, unless something goes wrong during exception handling. Namely two kind of exceptions may be raised: *internal exceptions* E which can be handled locally and *external exceptions* ϵ that must be signalled to the environment (the enclosing action or the whole system). Disjoint subsets of participants may join nested CAs and, consequently, nested exception contexts. Exceptions can be propagated along chains of nested actions: if the local handling of an exception E is not successful, then a corresponding exception ϵ will be thrown to the enclosing action. When one or more exceptions are raised in a CA the following actions are performed: (i) the cooperating threads are informed, (ii) nested actions are aborted because E has been thrown outside them, (iii) during abortion the handler may signal other exceptions, (iv) an algorithm determines which exception must be covered.

CAs have been adapted in [14] to model fault tolerant Web Services: the resulting Web Service Composition Actions (WSCA) relax transactional requirements over external objects since they cannot always be enforced in open systems. In the case of web services these transactional properties can be abstracted and left optional in the various services.

We model the cooperating threads in a CA as a set of participants $\{\mathtt{p}\}_{\{i \in I\}}$ performing a [*Link*]: the established session represents the outermost CA while nested CAs are implemented by try-catch blocks involving only a subset of threads/channels. For each (nested) CA we implement of an “exception resolver“ participant. This process is inactive during normal execution, but when one or more exceptions are thrown it collects the corresponding messages from the cooperating threads, decides which exception must be covered, and then sends the corresponding label to all participants. In the following we assume the resolver uses channel 1.

Let $\{E_h\}_{h\in H}$ and $\{\epsilon_k\}_{k\in K}$ be sets of internal and external exception respectively and

$$(\nu\tilde{s}')\begin{pmatrix} ...\mid \mathsf{try}(\tilde{s})\{\mathbf{0}\}\ \mathsf{catch}\ \{Q^1\} & \mid \mathsf{try}(\tilde{s})\{P^2\}\ \mathsf{catch}\ \{Q^2\}\mid \\ ...\mid \mathsf{try}(\tilde{s})\{P^n\}\ \mathsf{catch}\ \{Q^n\} & \mid s_1 : \varnothing \mid ... \mid s_m : \varnothing \end{pmatrix}$$

represent a CA where $\tilde{s} \subset \tilde{s}' = \{s_1, ...s_m\}$: $\tilde{s}'$ is the set of channels involved in the enclosing action while $\tilde{s}$ are the channels involved in the nested one (remember that nested actions, i.e. nested try-catch blocks can only involve proper subsets of channels).
The handler of the resolver process is:

$$Q^1 = \mathsf{try}(\tilde{s})\{s_1?(x_2).\dots s_1?(x_n).\}\ \mathsf{catch}\ \{\mathbf{0}\};\ Q^1_G \quad (1)$$
$$\text{where } Q^1_G = \mathsf{try}(\tilde{s})\{\text{ algorithm determining } h \in H.\ \ s_1!\langle(h)\rangle\dots s_1!\langle(h)\rangle\}\ \mathsf{catch}\ \{\mathbf{0}\} \quad (2)$$

The handlers Q^j for $j \in \{2, \dots, n\}$ have the shape:

$$Q^j{=}\mathsf{try}(\tilde{s})\{\mathsf{if}\ \mathtt{test}\ \mathsf{then}\ s_1!\langle h\rangle\ \mathsf{else}\ \ s_1!\langle 0\rangle\}\ \mathsf{catch}\ \{\mathbf{0}\};\ Q^j_G \quad (3)$$
$$\text{and } Q^j_G{=}\mathsf{try}(\tilde{s})\{s_1?(x).;P^j\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}')\}, \quad (4)$$

where `test` checks whether exception h has been raised by the current component.

Production Cell. We now implement a part of a case study modelling an industrial Production Cell. This example was proposed as a challenging case study by the FZI in 1993 [12]. The production cell consists of some devices (belts, elevating rotary table, press and rotary robot with two orthogonal extensible arms) associated with a set of sensors, and its task is to get a metal plate from its "environment" via the feed belt, transform it into the forged plate by using a press, and return it to the environment via the deposit belt. For a detailed explanation of the model see [16]. Here we just model the part of the system responsible of removing the plate from the press. The components we are considering are `Robot`, `RobotSensor`, `Press` and `PressSensor` abbreviated respectively as *R, RS, P, PS*. They are cooperating in the `remove-plate` action. There are two nested actions: `turn-robot-and-extend-arm` abbreviated as `TR` and `grab-plate-from-press` abbreviated as `GP`. For the sake of simplicity we assume that the exceptions that can be raised are `Robot-failure`, `Robot-sensor-failure`, `Press-failure`, `Press-Sensor-failure` abbreviated respectively as *RF, RSF, PF, PSF*. The handlers Q are indexed by the participants: Q^{RS} corresponds to the handler associated to the `robot-sensor`. In case of problems during exception handling the control is passed to the enclosing action by signalling one of the following exceptions: `BadRobotRecovery`, `BadRobotSensorRecovery`, `BadPressRecovery` and `BadPressSensorRecovery` abbreviated respectively as *BR, BRS, BP, BPS*.

The enclosing action `remove-plate` uses channels s_1, s_2. Concerning nested actions channel s_1 is used in action `turn-robot-and-extend-arm` while channel s_2 is used in action `grab-plate-from-press`. We recall that we write s as a shorthand for s^0. The action `remove-plate` is then implemented as in the following (where we omit dummy try-catch):

$$\mathtt{ResolverTR}\mid\mathtt{Robot}\mid\mathtt{RobotSensor}\mid\mathtt{Press}\mid\mathtt{PressSensor}\mid\mathtt{ResolverGP}\mid s_1 : L_1 \mid s_2 : L_2 \mid s_3 : L_3$$

where

$$\begin{array}{l}
\mathtt{ResolverTR} = \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{\mathbf{0}\}\ \mathsf{catch}\ \{R^{TR}\}\}\ \mathsf{catch}\ \{\mathbf{0}\}\\
\mathtt{ResolverGP} = \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{\mathbf{0}\}\ \mathsf{catch}\ \{R^{GP}\}\}\ \mathsf{catch}\ \{\mathbf{0}\}\\
\mathtt{Robot} = \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P^R\}\ \mathsf{catch}\ \{Q^R\}\}\ \mathsf{catch}\ \{Q'^R\}\\
\mathtt{RobotSensor} = \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{P^{RS}\}\ \mathsf{catch}\ \{Q^{RS}\};\mathsf{try}(s_2)\{P'^{RS}\}\ \mathsf{catch}\ \{Q'^{RS}\}\}\ \mathsf{catch}\ \{Q''^{RS}\}\\
\mathtt{Press} = \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_2)\{P^P\}\ \mathsf{catch}\ \{Q^P\}\}\ \mathsf{catch}\ \{Q'^P\}\\
\mathtt{PressSensor} = \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_2)\{P^S\}\ \mathsf{catch}\ \{Q^{PS}\}\}\ \mathsf{catch}\ \{Q'^{PS}\}.
\end{array}$$

Let us notice that each nested action has a corresponding resolver. Let us focus on `Robot | RobotSensor` (where we put $P' = \mathsf{try}(s_2)\{P'_{RS}\}\ \mathsf{catch}\ \{Q'^{RS}\}$) and suppose there is a failure in the `Robot` and concurrently in the `Robot-sensor`, that is:

$$\begin{array}{l}
\mathtt{ResolverTR}\mid\mathtt{Robot}\mid\mathtt{RobotSensor} \longrightarrow^*\\
\mathtt{ResolverTR}\mid \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{\mathsf{throw}(s_1);P''\}\ \mathsf{catch}\ \{Q^R\}\}\ \mathsf{catch}\ \{Q'^R\}\mid\\
\mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1)\{\mathsf{throw}(s_1);P'''\}\ \mathsf{catch}\ \{Q^{RS}\};P'\}\ \mathsf{catch}\ \{Q'^{RS}\}
\end{array}$$

We apply rule $\lfloor$Thr$\rfloor$ and $\lfloor$RThr$\rfloor$ twice obtaining:

$$\begin{array}{ll}
\mathsf{throw}(s_1) \vdash & \mathtt{ResolverTR}\mid\ \mathsf{try}(s_1,s_2)\{Q^R\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q'^R\}\mid\\
& \mathsf{try}(s_1,s_2)\{Q^{RS}\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q'^{RS}\}
\end{array}$$

After the substitution $\{s_1^1/s_1\}$ we have:

$$\begin{array}{ll} \mathsf{throw}(s_1) \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{s_1^1?(x_1);\, s_1^1?(x_2)\}\ \mathsf{catch}\ \{\mathbf{0}\};\, Q_G\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{\mathbf{0}\} \mid \\ & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathsf{if}\ldots\}\ \mathsf{catch}\ \{\mathbf{0}\};\, Q_G^R\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q'^R\} \mid \\ & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathsf{if}\ldots\}\ \mathsf{catch}\ \{\mathbf{0}\};\, Q_G^{RS}\{s_1^1/s_1\};\, P'\}\ \mathsf{catch}\ \{Q'^{RS}\} \mid s_1[1] : (\emptyset), \end{array}$$

we apply both rule ⌊Send⌋ and rule ⌊Rec⌋ twice:

$$\begin{array}{ll} \mathsf{throw}(s_1) \vdash & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathbf{0}\};\, Q_G\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{\mathbf{0}\} \mid \\ & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathbf{0}\};\, Q_G^R\{s_1^1/s_1\}\}\ \mathsf{catch}\ \{Q'^R\} \mid \\ & \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathbf{0}\};\, Q_G^{RS}\{s_1^1/s_1\};\, P'\}\ \mathsf{catch}\ \{Q^{RS}\} \mid s_1[1] : (\emptyset). \end{array}$$

Now the execution goes on with the general handlers processes:

$$\begin{array}{l} \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\text{ algorithm determining } h \in H.\ \ s_1!\langle(h)\rangle\}\ \mathsf{catch}\ \{\mathbf{0}\}\}\ \mathsf{catch}\ \{\mathbf{0}\} \mid \\ \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{s_1?((x));\ \text{informing external objects}\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1^1,s_2)\}\}\ \mathsf{catch}\ \{Q^R\} \mid \\ \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{s_1?((x));\ \text{informing external objects};\, P'\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1^1,s_2)\}\}\ \mathsf{catch}\ \{Q^{RS}\} \mid s_1[1] : (\emptyset), \end{array}$$

the resolver reads the value of the received labels, calculates which exception must be covered and sends the corresponding label to the other processes. As explained above the handler of Q_G's processes is a throw on the outer set of channels: the reason is that if an exception is raised during the general handler execution, the exception must be recovered by the enclosing action. Now there are two cases:

1. the execution of the general handlers terminates without problems. In this case the execution goes on with the next nested action `grab-press-from-plate` in which `RobotSensor`, `Press` and `PressSensor` cooperate:

   $$\begin{array}{l} \mathsf{throw}(s_1) \vdash \mathsf{try}(s_1,s_2)\{\mathbf{0}\}\ \mathsf{catch}\ \{Q'^R\} \mid \mathsf{try}(s_1,s_2)\{P'\}\ \mathsf{catch}\ \{Q'^{RS}\} \\ \mathtt{Press} \mid \mathtt{PressSensor} \mid s_1 : \ldots \end{array}$$

2. something goes wrong during the general handlers execution (for instance $Q_G^R \longrightarrow^* \mathsf{throw}(s_1^1)$). Then the corresponding handler alerts the enclosing action by signalling a throw on channels s_1^1, s_2:

   $$\begin{array}{l} \mathsf{throw}(s_1) \vdash \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathsf{throw}(s_1^1)\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1^1,s_2)\}\}\ \mathsf{catch}\ \{Q'^R\} \mid \\ \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{Q_G^{RS}\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1^1,s_2)\};\, P'\}\ \mathsf{catch}\ \{Q'^{RS}\}, \end{array}$$

   we apply rule ⌊Thr⌋:

   $$\begin{array}{l} \mathsf{throw}(s_1), \mathsf{throw}(s_1^1) \vdash \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1^1,s_2)\}\}\ \mathsf{catch}\ \{Q'^R\} \mid \\ \mathsf{try}(s_1,s_2)\{\mathsf{try}(s_1^1)\{Q_G^{RS}\}\ \mathsf{catch}\ \{\mathsf{throw}(s_1^1,s_2)\};\, P'\}\ \mathsf{catch}\ \{Q'^{RS}\}, \end{array}$$

   we apply rule ⌊RThr⌋ twice:

   $$\begin{array}{l} \mathsf{throw}(s_1), \mathsf{throw}(s_1^1) \vdash \mathsf{try}(s_1,s_2)\{\mathsf{throw}(s_1^1,s_2)\}\ \mathsf{catch}\ \{Q'^R\} \mid \\ \mathsf{try}(s_1,s_2)\{\mathsf{throw}(s_1^1,s_2);\, P'\}\ \mathsf{catch}\ \{Q'^{RS}\} \end{array}$$

   we apply rule ⌊Thr⌋ twice with:

   $$\begin{array}{l} \mathsf{throw}(\natural(s_1^1,s_2)) = \mathsf{throw}(s_1,s_2) \\ \mathsf{throw}(s_1), \mathsf{throw}(s_1^1), \mathsf{throw}(s_1,s_2) \vdash \mathsf{try}(s_1,s_2)\{\mathbf{0}\}\ \mathsf{catch}\ \{Q'^R\} \mid \mathsf{try}(s_1,s_2)\{P'\}\ \mathsf{catch}\ \{Q'^{RS}\}, \end{array}$$

   Coming back to the complete action `remove-plate`:

   $$\begin{array}{l} \mathsf{throw}(s_1), \mathsf{throw}(s_1^1), \mathsf{throw}(s_1,s_2) \vdash Q^R\{s_1^2, s_2^1/s_1^1, s_2\} \mid Q^{RS}\{s_1^2, s_2^1/s_1^1, s_2\} \mid \\ Q^P\{s_1^2, s_2^1/s_1^1, s_2\} \mid Q^{PS}\{s_1^2, s_2^1/s_1^1, s_2\}. \end{array}$$

   In this case the execution goes on handling an external exception $i \in \{BR, BRS, BP, BPS\}$. Let us notice that the following nested action, `grab-plate-from-press`, is not executed because of a failure involving the enclosing action.

Correctness and complexity. Let N the number of interacting participants, T_{nmax} be the maximum time of message passing between participants, T_{reso} be the upper bound of the time spent in resolving current exceptions, T_{abort} be the maximum possible time for a thread to abort one nested CA, T_{throw} the cost for signalling the throw (namely to put it in the Σ), $nmax$ be the maximum number of nesting levels of CAs (if no nesting, then $nmax = 0$), Δ_{nmax} be maximum possible time of handling an (resolving) exception. We share with [16] the following results:

1. Any participant p, will complete exception handling ultimately in at most T, where $T = (nmax + 3)T_{nmax} + nmax \cdot T_{abort} + (nmax + 1)(T_{reso} + \Delta_{nmax}) + T_{throw}$.
2. For a given CA A, if no exception is raised in any enclosing action of A, then no more new exceptions will be raised within A once the exception resolution starts.
3. If multiple exceptions are raised concurrently, an ultimate resolving exception that covers all the exceptions will be generated by the proposed algorithm.
4. The number of messages is independent of the number of concurrent exceptions. Taking the nesting of actions into account, in the worst case, our approach requires exactly $nmax(N - 1)$ messages+ N throws. (Let us notice that in [16] the algorithm performs in $O(N^2)$ messages).

6 Conclusions

We have introduced a type-safe global escape mechanism for handling unexpected or unwanted conditions changing the default execution of distributed communicational flows, by means of a collection of asynchronous local exceptions. All the involved conversation parties are guaranteed of the communication safety even after an unforeseen event has been encountered. We have defined a calculus and a type discipline based on the multiparty session [9], and show that the multiparty session types provide a rigorous discipline which can describe and validate complex exception scenarios such as criss-crossing global interactions and fault tolerance for distributed cooperating threads. The flexibility was actually realised allowing local exceptions to be thrown at any stage of the conversation and to any subset of participants. Concerning the criss-cross example our implementation of the protocol never moves to the situation where the Seller sends a confirmation to the Client but the Client aborts the interaction or the Client accepts the wrong loan offer from the Bank.

Related work. Exception handling has been studied for many programming languages including communication-based ones: in distributed object-oriented programming [13, 16], in particular [16] presents the algorithm we implemented as an example in Section 5; several service-oriented calculi (e.g. [2, 11, 15]) include mechanisms for compensation or termination handling, but none of those mechanisms provide a means for coordinating all involved peers that move together to a new stage of the conversation when the unexpected condition is encountered. The paper [10] compares the expressive power of different approaches to compensation; w.r.t. their classification our approach has a static compensation definition, is nested, and has no protection operator. In the context of session types theory, [4, 6] proposed *interactional* exceptions, which inspired our work, for binary sessions and for web service choreographies. The approach described in [4, 6] is significantly different from ours, due to the fact that: (i) exceptions are modelled as special messages exchanged by the parties; (ii) try-catch blocks cannot be at any point in the program but only after a session connection: this means that for a conversation a default behaviour and an exceptional one are defined, while in our calculus try-catch blocks can occur at any point, even nested; and (iii) in those calculi nested try-catch blocks come from nested session connections, and inner exception handlers are refinements of outer ones, while in our case nested try-catch blocks always belong to the same conversation (we forbid session connections inside a try block) and inner exceptions always involve less peers than outer exceptions.

Future work. For the sake of simplicity, so far we have not included in the calculus an important mechanism in the context of session types: session delegation. Even if session delegation seems to be less interesting in the multiparty sessions context than in the binary sessions one, because of the presence of several participants instead of just two, we believe this mechanism is worth of further investigation. Another feature that seems promising w.r.t. practical examples is the capability of distinguishing among different kinds of exceptions (with corresponding different kinds of handlers): the calculus can be easily extended in this direction by putting the right constraints (i.e. all participants

must be able to handle the same set of exceptions). Finally we plan to integrate with multiparty logic work [1] by which we can write a wide range of global escape scenarios which require fine-grained behavioural specifications given by logical assertions, and still ensure communication safety and progress.

Acknowledgments. We thank the FSTTCS reviewers for useful comments. The members of WS-CDL (http://www.w3.org/2002/ws/chor/) and Scribble (http://www.jboss.org/scribble) (in particular, Gary Brown, Kohei Honda and Nickolas Kavantzas) provided many use cases which motivated us to study this subject: we used some of them as the main examples of this paper.

References

1. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010.
2. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS 2003*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.
3. S. Capecchi, E. Giachino, and N. Yoshida. Global Escape in Multiparty Sessions. Technical report, Imperial College, London, GB, 2010.
4. M. Carbone, K. Honda, and N. Yoshida. Structured interactional exceptions for session types. In *CONCUR 2008*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008.
5. M. Carbone, K. Honda, and N. Yoshida. Theoretical aspects of communication-centred programming. *ENTCS*, 209:125–133, 2008.
6. Marco Carbone. Session-based choreography with exceptions. *ENTCS*, 241:35–55, 2009.
7. P. Deniélou and N. Yoshida. Buffered communication analysis in distributed multiparty sessions. In *CONCUR 2010*, volume 6269 of *LNCS*, pages 343–357. Springer, 2010.
8. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
9. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. volume 43, pages 273–284, New York, NY, USA, 2008. ACM.
10. I. Lanese, C. Vaz, and C. Ferreira. On the expressive power of primitives for compensation handling. In *ESOP 2010*, volume 6012 of *LNCS*, pages 366–386. Springer, 2010.
11. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
12. C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, London, UK, 1995. Springer-Verlag.
13. C. M. F. Rubira and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95*, page 499, Washington, DC, USA, 1995. IEEE Computer Society.
14. F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated forward error recovery for compositeweb services. *Reliable Distributed Systems, IEEE Symposium on*, 0:167, 2003.
15. H. Torres Vieira, L. Caires, and J. Costa Seco. The conversation calculus: A model of service-oriented computation. In *ESOP 2008*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.
16. J. Xu, A. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: From model to system implementation. In *ICDCS '98*, pages 12–21, Washington, DC, USA, 1998. IEEE Computer Society.