Structured Interactional Exceptions in Session Types

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

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

Abstract. We propose an interactional generalisation of structured exceptions based on the session type discipline. Interactional exceptions allow communicating peers to asynchronously and collaboratively escape from the middle of a dialogue and reach another in a coordinated fashion, under an arbitrary nesting of exceptions. New exception types guarantee communication safety and offer a precise type-abstraction of advanced conversation patterns found in practice. Protocols for coordinating normal and exceptional exit among asynchronously running sessions are introduced. The liveness property established under these protocols guarantees consistency of coordinated exception handling among communicating peers.

1 Introduction

Structured exceptions in modern programming languages such as Java and C♯ allow a thread of control in a block (often designated as "`try` block") to get transferred to another block (exception handler, "`catch` block"), when a system or user raises an event called *exception*. Their central merit is to enable a dynamic escape from a block of code to another (like `goto`), but in a controlled and structured way (unlike `goto`). They are useful not only for error-handling but more generally for a flexible control flow while preserving well-structured description and type-safety.

This paper studies the new notion of structured exceptions for distributed, concurrent, asynchronously communicating programs based on session types [10, 17], motivated by collaboration with industry partners in web services [19] and financial protocols [14]. These two application domains contain a wealth of structured conversation patterns arising from practical needs [11], and many of these patterns crucially rely on dynamic escape: a conversation is interrupted by a special communication action, after which all peers move to a different stage. Realising such conversation patterns requires a consistent propagation of exception messages among concurrently communicating peers; an exception affects not only a sequential thread but also a collection of parallel processes; and an escape needs to move into another dialogue in a concerted manner. The distinguishing feature of these exceptions in comparison with their traditional counterpart is that they demand not only local but also coordinated actions among communicating peers. We call such exceptions, *interactional exceptions*.

As a simple example of interactional exceptions, we present the following scenario, coming from widely used financial protocols. Henceforth we assume a message passing in a session is asynchronous, i.e. the completion of a sending action does not need a handshake with its receiver, a standard assumption in financial messaging [1]. Suppose Seller wishes to sell a product to Buyer.

1. Seller repeats sending quotes of a product without waiting for an acknowledgement;
2. When Buyer replies with his interest in one of the quotes, the loop terminates and Seller and Buyer move to another stage, for e.g. completion of the transaction.

This simple conversation pattern contains an asynchronous escape from one part of a conversation to another. After one party aborts, the same thing should happen to the other, both moving together to another part of the conversation.

As a second example, we continue the above scenario, extending to the situation where Buyer and Seller negotiate the price of the product through Broker.

1. Buyer initiates a conversation (session) with Broker, in order to buy a product.
2. As a result, Broker initiates a conversation with Seller, and starts brokering between Buyer and Seller, to reach a successful transaction.
3. If an exceptional circumstance arises between 1 and 2 (e.g. a legal issue), Buyer or Broker will abort and they together move to an exception dialogue to quit the transactions formally.
4. On the other hand, if there is an exceptional circumstance during 2, then there is an exception dialogue involving all of Broker, Seller and Buyer.

Above, an exception handling at Broker is *nested*, whose later, or inner, exception handling (4, involving all three parties) supersedes the earlier, or outer, one (3, involving only Broker and Seller). As a conversation evolves, more communication peers may be involved, making it necessary to coordinate more parties when an exception is raised.

To maintain the virtues of traditional structured exceptions, as well as those of the existing session types discipline, we may as well demand the following three properties for this generalised form of exceptions.

- *flexibility*: it should allow asynchronous escape at any desired point of a conversation, including nested exceptions;
- *consistency*: even under asynchrony, messages in a "default" conversation should not get mixed up with those in an "exception" conversation, under arbitrary nesting;
- *type safety*: communications inside a session always take place linearly and without communication mismatch, carrying out fundamental properties of foregoing session type disciplines.

We address these requirements by extending session types with the following features:

1. Asynchronous exceptions where nested scopes are consistently handled by a meta-level reduction and a stack discipline. A simple machinery, based on exception levels, prevents mix-up of messages in normal and exception conversations.
2. An operational structure for coordinating exceptions including the protocols to propagate exceptions, to handle normal and exceptional exit from a conversation, and to coordinate entry into exception-handling conversations.
3. A type structure for interactional exceptions which minimally extends that of the existing session types. They ensure communication safety and liveness, which together guarantee the consistency of the introduced operational structures.

The stipulated formal semantics of interactional exceptions is intended to suggest a possible framework of implementation, as discussed in § 5. As far as we know, this work is the first to present a consistent extension of the session types discipline to interactional exceptions, backed up by its key formal properties.

2 Session Calculus with Interactional Exceptions

Syntax. We introduce the syntax of processes using the π-calculus with session primitives [10]. Let a, b range over *service channels*; s, r, t over *session channels*; x, y, z over *variables*; and $X, X', \ldots$ over *term variables*. We first introduce the syntax of processes ($P, Q, R, \ldots$) written by programmers.

$$
\begin{array}{rlrl}
P ::= & *\, c(\lambda)[P, Q] & \text{(service)} & \mid \overline{c}(\lambda)[\tilde{\kappa}, P, Q] \quad \text{(request)} \\
\mid & \kappa?(x).\, P & \text{(input)} & \mid \kappa!\langle e \rangle.\, P \quad \text{(output)} \\
\mid & \kappa \triangleright \{l_i : P_i\}_{i \in I} & \text{(branch)} & \mid \kappa \triangleleft l.\, P \quad \text{(select)} \\
\mid & P \mid Q & \text{(par)} & \mid \textbf{if } e \textbf{ then } P \textbf{ else } P \quad \text{(cond)} \\
\mid & \mathbf{0} & \text{(inact)} & \mid (\nu a)\, P \quad \text{(resServ)} \\
\mid & X & \text{(termVar)} & \mid \mu X.\, P \quad \text{(recursion)} \\
\mid & \textbf{throw} & \text{(throw)} &
\end{array}
$$

$$
e ::= a \mid \mathsf{tt} \mid \mathsf{ff} \mid e \,\mathsf{and}\, e \mid \neg e \mid \ldots \qquad c ::= a \mid x \qquad \kappa, \lambda ::= s^p
$$

Session channel s^p is a *polarised* channel [8] where variable p ranges over polarities $\{+, -\}$. We define the dual of a polarised channel s^p as $\overline{s^+} = s^-$ and $\overline{s^-} = s^+$.

A service $*a(\lambda)[P, Q]$, named a, is a replicated process where P is the *default process* and Q is the *exception handler*. By replication a service is always available (following Service Channel Principle (SCP): "services should always be available in multiple copies" [6], like services at URLs). When $*a(\lambda)[P, Q]$ is requested for a session via a shared name a, a fresh session channel is established, and through this channel P is engaged in a series of communication actions, possibly followed by Q if an exception takes place. A request $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$ interacts with a service via c and establishes a fresh session λ, with its *default process* P and *handler* Q. The channels $\tilde{\kappa}$ are the already established sessions with which the handler Q gets associated with, thus allowing nesting of exceptions. We also let λ itself be included in $\tilde{\kappa}$, which is convenient for typing. We call $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$ a *refinement* of each κ_i for $\kappa_i \in \{\tilde{\kappa}\} \setminus \{\lambda\}$, since its handler Q refines the handlers of the previous sessions enabling nested exceptions. **throw** is a process which is about to throw an exception. All other constructs are from [6, 10].

Free/bound (term) variables/channels and α-equivalence are standard. $\mathsf{fsc}(P)$, $\mathsf{fn}(P)$ and $\mathsf{fv}(P)$ respectively denote the sets of free session channels, service channels, and variables in P. We call *program* a process which does not contain free variables or free session channels. We often omit the tailing $\mathbf{0}$.

For having consistent operational semantics, we stipulate the following syntactic constraints: (i) recursions should be *guarded*, i.e. P in $\mu X.\, P$ is prefixed by an input/output/branch/select/conditional; (ii) a service can never occur under an input/output/recursion prefix nor inside a default process or a handler thus protecting the availability of services from exceptions; and (iii) in $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$, a free term variable never occurs in P or Q and, for each $\overline{c'}(\lambda')[\tilde{\kappa}', P', Q']$ occurring in P, we have $\kappa_i \in \tilde{\kappa}'$ if and only if $\tilde{\kappa} \subseteq \tilde{\kappa}'$ (for consistent exception propagation). Further, such a refinement never occurs inside a handler (otherwise we have ambiguity when launching a handler).

In the present paper we also stipulate that **throw** never occurs inside a handler. This prevents a handler from throwing a further exception in the same session. We do not consider such "cascading exceptions" (which is another kind of nested exceptions) for the sake of simpler presentation: their treatment is discussed in §5.

Example 1 (Asynchronous Escape) We can write the first example in § 1 as:

$$\begin{array}{ll} \texttt{Buyer} = \overline{\texttt{chSeller}}(s^+)[\ s^+, & \texttt{Seller} = *\texttt{chSeller}(s^-)[\\ \quad \mu X.\ s^+?(y).\ \textbf{if}\ \texttt{ok}(y)\ \textbf{throw else}\ X, & \quad \mu X.\ s^-!\langle \texttt{quote} \rangle.\ X\ , \\ \quad s^+!\langle \texttt{card} \rangle.\ s^+?(z)\ ] & \quad s^-?(y_2).\ s^-!\langle \texttt{time} \rangle\ ] \end{array}$$

Buyer keeps on reading messages on s^+ until condition $\texttt{ok}(y)$ is met and then it throws an exception. Seller, instead, is in an infinite loop where it persistently sends a quote over channel s^- (we assume `quote` changes over time). When the exception is raised the handlers are run: Buyer will send the credit card details `card` and Seller will acknowledge on channel s^- with the current time.

Example 2 (Nested Escapes) The second example given in the introduction, can be represented in our calculus as (Seller is unchanged from Example 1):

$$\begin{array}{ll} \texttt{Buyer} = \overline{\texttt{chBroker}}(t^+)[\ t^+, & \texttt{Broker} = *\texttt{chBroker}(t^-)[\ t^-, \\ \quad t^+!\langle \texttt{id} \rangle. & \quad t^-?(x).\ \textbf{if}\ \texttt{bad}(x)\ \textbf{then throw else} \\ & \quad \overline{\texttt{chSeller}}(s^+)[\ (s^+, t^-), \\ \quad \mu X.\ t^+?(y).\ \textbf{if}\ \texttt{ok}(y)\ \textbf{throw else}\ X, & \quad\quad \mu X.\ s^+?(x).\ t^-!\langle x + 10\% \rangle.\ X, \\ & \quad\quad t^- \triangleleft l_1.\ t^-?(y_2).\ s^+!\langle y_2 \rangle. \\ \quad t^+ \triangleright \{\ l_1 : t^+!\langle \texttt{card} \rangle.\ t^+?(z), & \quad\quad s^+?(y_3).\ t^-!\langle y_3 \rangle \quad], \\ \quad\quad l_2 : P_{\texttt{abort}}\} \quad] & \quad t^- \triangleleft l_2.\ R_{\texttt{abort}} \quad] \end{array}$$

Buyer first sends its identity `id` and then Broker throws an exception or proceeds by invoking Seller based on `bad(id)`. In the first case, process $t^- \triangleleft l_2.\ R_{\texttt{abort}}$ in the outermost handler selects the l_2 branch on Buyer's handler and proceeds with abortion (conversation between $P_{\texttt{abort}}$ and $R_{\texttt{abort}}$). In the other case, Seller is invoked and the protocol proceeds as in Example 1 with Broker forwarding messages and increasing quotes by 10%. When Buyer decides to accept a quote, the innermost handler is run by Broker which selects the l_1 conversation in Buyer's handler and forwards the exception to Seller. Then Broker forwards messages, successfully completing the transaction.

Semantics. We augment the semantics of asynchronous sessions [4, 9, 12] with exception handling (i.e. shutting down a default process and launching the corresponding handler) and exception propagation (informing session peers of an exception occurrence, realised by propagation of the special *exception message* †). Further we ensure that processes always carry out their conversation at properly matching levels (for example when a default process sends a message, a receiving peer may throw an exception before the message arrives, making it no longer relevant), by annotating message queues, hence in effect messages in them, with exception levels.

We use the following *runtime processes* [4, 9, 12] to define the operational semantics, extending the grammar of programs.

$$\begin{array}{lll} P ::= \ \ldots \mid (\nu s)\ P & \text{(resSess)} & \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \qquad \text{(queue)} \\ \quad \mid \textbf{try}\{\ P\ \}\ \textbf{catch}\ \{\ \tilde{\kappa} : Q\ \} & \text{(try-catch)} & \mid \tilde{\kappa}[\![P]\!] \qquad \text{(wrap)} \\ L ::= \quad \epsilon \mid h :: L & & h ::= \quad l \mid a \mid \ \texttt{tt} \ \mid \ \texttt{ff} \ \mid \dagger \end{array}$$

Free variables and channels are extended to run-time processes. Session restriction $(\nu s)\, P$ is standard. For formalising order-preserving asynchronous message passing, we use a directed message queue $\kappa \hookrightarrow_\phi \overline{\kappa} : L$ [4, 9], where κ (source) and $\overline{\kappa}$ (target) are two dual endpoint session channels. ϕ ranges over natural numbers, describing the level of the exception at which messages in the queue are to be received, relative to the current position of the queue (we do not need to consider the level of a sender, since this level is recorded by the number of the exception messages † inside a queue). We often write $\kappa \hookrightarrow \overline{\kappa} : L$ for $\kappa \hookrightarrow_0 \overline{\kappa} : L$. The list $L :: h$ is obtained by extending L with an extra tail element h. The *try-catch block* $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\}$ is the runtime presentation of a default process and a handler: the default process P in the *try-block* is running during which an exception on channels $\tilde{\kappa}$ can be thrown, which terminates P and launches the handler Q in the *catch-block*. When this Q is launched, it becomes a *wrapped process* or a *wrap*, $\tilde{\kappa}[\![Q]\!]$, making Q immune to an exception notification at the same or upper levels (note such notifications can come due to asynchrony). The transition from a try-catch to a wrap is realised by the meta reduction.

Meta Reduction. The *meta reduction* (1) erases the remaining activity of the default process in the try-block; (2) propagates exceptions to the try-catch blocks inside the try-block; and (3) leaves wrapped processes as they are. In traditional structured exceptions as found in Java or C++, an exception completely erases the try-block and lets the handler run in the same state. In our calculus, concurrently running threads inside a try-block may have conversations (sessions) with other agents. Erasing them would make conversations inconsistent, thus an exception is thrown in each of them.

The meta reduction is written $P \searrow (P', S)$, where the initial process P is transformed into process P', the result of erasing and wrapping; and S denotes session channels via which we should communicate that the exception takes place including the ones of nested try-catch blocks. The rules are defined as follows

$$(\text{MTry})\quad P \searrow (P', S) \quad \Rightarrow \quad \mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \searrow \begin{cases} (P', S) & \text{if } \tilde{\kappa} \subseteq S \\ (\tilde{\kappa}[\![Q]\!] \mid P', S \cup \tilde{\kappa}) & \text{otherwise} \end{cases}$$

$$(\text{MWrap})\quad \tilde{\kappa}[\![Q]\!] \searrow (\tilde{\kappa}[\![Q]\!], \emptyset)$$

$$(\text{MPar})\quad P \searrow (P', S_1) \text{ and } Q \searrow (Q', S_2) \quad \Rightarrow \quad P \mid Q \searrow (P' \mid Q', S_1 \cup S_2)$$

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

(MTry) propagates the exception to a nested try-catch block. If the try-block meta reduces to some P' with some set S then $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\}$ will reduce either to (i) P' itself or to (ii) the parallel composition of P' and $\tilde{\kappa}[\![Q]\!]$ with the new set $S \cup \tilde{\kappa}$ ensuring that also channels $\tilde{\kappa}$ will be notified with an exception. Case (i) discards handler Q when another handler for $\tilde{\kappa}$ is already in P while case (ii) happens when there is no refinement of $\tilde{\kappa}$ in P. The mechanism is sound because of the assumption that κ_i are always refined together (cf. syntax). Note that, if the try-block is single-threaded, the meta reduction mechanism is identical to the one of standard exception handling.

Reduction. We now introduce the main reduction rules. Due to the nesting of wraps and try-catch blocks, the reduction is defined using the following reduction context:

$$C \ ::= \mathbf{try}\{\, C \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \quad | \quad P \mid C \quad | \quad \tilde{\kappa}[\![C]\!] \quad | \quad (\nu s)\, C \quad | \quad (\nu a)\, C \quad | \quad -$$

Table 1 Reduction Semantics

(Init) $$*a(s^-)[P,Q] \mid C[\overline{a}(s^+)[\tilde{\kappa}, P', Q']] \longrightarrow *a(s^-)[P,Q] \mid (\nu s) \begin{pmatrix} \mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, s^- : Q \,\} & \mid & s^- \hookrightarrow_0 s^+ : \epsilon \mid \\ C[\mathbf{try}\{\, P' \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q' \,\}] & \mid & s^+ \hookrightarrow_0 s^- : \epsilon \end{pmatrix}$$

(Out) $$\kappa!\langle e \rangle.\, P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : (v :: L) \qquad (e \downarrow v)$$

(In) $$\kappa?(x).\, P \mid \overline{\kappa} \hookrightarrow_0 \kappa : (L :: v) \longrightarrow P\{v/x\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : L$$

(Sel) $$\kappa \triangleleft l.\, P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : L \longrightarrow P \mid \kappa \hookrightarrow_\phi \overline{\kappa} : (l :: L)$$

(Bra) $$\kappa \triangleright \{l_i : P_i\}_{i \in I} \mid \overline{\kappa} \hookrightarrow_0 \kappa : (L :: l_j) \longrightarrow P_j \mid \overline{\kappa} \hookrightarrow_0 \kappa : L \qquad (j \in I)$$

(Con) $$P \longrightarrow Q \quad \Rightarrow \quad C[P] \longrightarrow C[Q]$$

(If) $$\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow P \quad (e \downarrow \mathsf{tt}) \qquad \mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q \longrightarrow Q \quad (e \downarrow \mathsf{ff})$$

(Str) $$P \equiv P' \text{ and } P' \longrightarrow Q' \text{ and } Q' \equiv Q \quad \Rightarrow \quad P \longrightarrow Q$$

(Thr) $$\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \searrow (R,S) \quad \Rightarrow$$
$$\mathbf{try}\{\, \mathbf{throw} \mid P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$$

(RThr) $$\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \searrow (R,S) \quad \Rightarrow$$
$$\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q \,\} \mid \overline{\kappa}_j \hookrightarrow_0 \kappa_j : (L :: \dagger) \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \overline{\kappa}_j \hookrightarrow_1 \kappa_j : L \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$$

(WVal) $$\tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: v) \longrightarrow \tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : L$$

(WThr) $$\tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_0 \kappa_i : (L :: \dagger) \longrightarrow \tilde{\kappa}\{\!|Q|\!\} \mid \overline{\kappa}_i \hookrightarrow_1 \kappa_i : L$$

(Clean) $$P \searrow (R,S),\ (\lambda \in \tilde{\kappa}, \dagger \in L) \Rightarrow$$
$$\mathbf{try}\{\, P \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : \tilde{Q} \,\} \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : L_\kappa \longrightarrow R \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \mid \Pi_{\kappa \in S}\ \kappa \hookrightarrow_{\phi_\kappa} \overline{\kappa} : (\dagger :: L_\kappa)$$

The reduction $\longrightarrow$ is the smallest relation generated by the rules in Table 1. (Init) gives the semantics of session initiation, generating two fresh dual session channels, the associated two empty queues (ϵ denotes the empty string) and the two try-catch blocks $\mathbf{try}\{\, P \,\}\ \mathbf{catch}\ \{\, s^- : Q \,\}$ and $\mathbf{try}\{\, P' \,\}\ \mathbf{catch}\ \{\, \tilde{\kappa} : Q' \,\}$. Note that $*a(s^-)[P,Q]$ is not in a context. This is because we have assumed that *services never appear nested in a try- or a catch-block* as we do not want them to be terminated (following SCP).

(Out) and (Sel) enqueue, respectively, a value and a label at the head of the queue for κ. Symmetrically, (In) and (Bra) dequeue from the tail of the queue. The exception level in the latter two rules is 0, indicating the level of an actual receiver. The exception level of a queue ensures that a message is sent and received at the same level, guaranteeing consistency of communication. This depends on the invariance that the sum of the level of the queue and the number of †'s in the queue before a specific message, determines the depth (the number of wraps) at which the message enqueueing is performed. In (Out,If), $e \downarrow v$ says that expression e evaluates to value v. (Con,Str) are standard.

(Thr) and (RThr) represent the firing of an exception. (Thr) is when **throw** appears top-level in the try-block, i.e. exception is thrown locally; while (RThr) is when a remote exception is received as † in the queue. Eventually, all peers will be notified of the exception by sending † via channels in S generated from P as well as $\tilde{\kappa}$. An alternative semantics prioritises † [2].

Rule (WVAL) describes the case when messages at the default level meet a wrapped process and are drained into a sink (i.e. get dequeued but ignored). In (WTHR), † meets a wrap and the exception level of the queue is incremented, allowing the queue to enter the wrap. In (CLEAN), † in the queue reveals the presence of a refinement in P which has now become a wrap due to a local throw. Meta reduction propagates the exception to each parallel process in P and the try-catch block is discarded.

This last step is formally defined by the structural congruence $\equiv$ which plays a key role in treating exceptions and, in particular, moving queues while maintaining their exception levels. $\equiv$ is the least congruence relation on processes such that $(P, |)$ is a commutative monoid and includes the standard rules for restriction (such as scope extrusion) and recursion. In addition, it has the following rules:

a) $\mathbf{try}\{\, P \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \,\}\, \mathbf{catch}\, \{\, \tilde{\kappa} : Q \,\} \;\equiv\; \mathbf{try}\{\, P \,\}\, \mathbf{catch}\, \{\, \tilde{\kappa} : Q \,\} \mid \lambda \hookrightarrow_\phi \overline{\lambda} : L \quad (\lambda \in \tilde{\kappa} \Rightarrow \dagger \notin L)$

b) $\tilde{\kappa}[\![P \mid \overline{\lambda} \hookrightarrow_\phi \lambda : L]\!] \;\equiv\; \tilde{\kappa}[\![P]\!] \mid \overline{\lambda} \hookrightarrow_\phi \lambda : L \qquad (\lambda \notin \tilde{\kappa})$

c) $\tilde{\kappa}[\![P]\!] \mid \overline{\kappa}_i \hookrightarrow_\phi \kappa_i : L \;\equiv\; \tilde{\kappa}[\![P \mid \overline{\kappa}_i \hookrightarrow_{\phi-1} \kappa_i : L]\!]$

d) $\mathbf{try}\{\, (\nu a)\, P \,\}\, \mathbf{catch}\, \{\, \tilde{\kappa} : Q \,\} \;\equiv\; (\nu a)\, \mathbf{try}\{\, P \,\}\, \mathbf{catch}\, \{\, \tilde{\kappa} : Q \,\} \qquad (a \notin \mathsf{fn}(Q))$

e) $\tilde{\kappa}[\![(\nu a)\, P]\!] \;\equiv\; (\nu a)\, \tilde{\kappa}[\![P]\!]$

The first and second rules allow a queue to move into a try-catch block and a wrap respectively. The third rule is applicable when the receiving side of the queue is in $\tilde{\kappa}$: when entering the wrap, ϕ is decreased so that the process inside the wrap can read the value if the level after the decrement is 0. The last two rules open the scope.

To illustrate how queue levels work, we consider the following process:

$$\begin{aligned} P = \mathbf{try}\{\ \mathbf{throw} \mid \kappa!\langle 5\rangle\ \}\, \mathbf{catch}\, \{\, \kappa : \kappa!\langle \mathtt{tt}\rangle \,\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : \epsilon \mid \\ \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\, \mathbf{catch}\, \{\, \overline{\kappa} : \overline{\kappa}?(x) \,\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \end{aligned}$$

Process P can reduce to $P' = \kappa[\![\mathbf{0}]\!] \mid \overline{\kappa}[\![\mathbf{0}]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \epsilon \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon$ in different ways.

$$\begin{aligned} P &\longrightarrow\equiv \kappa[\![\kappa!\langle \mathtt{tt}\rangle]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger \mid \mathbf{try}\{\ \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\, \mathbf{catch}\, \{\, \overline{\kappa} : \overline{\kappa}?(x) \,\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \\ &\longrightarrow\equiv \kappa[\![\mathbf{0}]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : (\mathtt{tt} :: \dagger) \mid \mathbf{try}\{\, \mathbf{throw} \mid \overline{\kappa}?(x)\ \}\, \mathbf{catch}\, \{\, \overline{\kappa} : \overline{\kappa}?(x) \,\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \epsilon \\ &\longrightarrow\equiv \kappa[\![\mathbf{0}]\!] \mid \kappa \hookrightarrow_1 \overline{\kappa} : \mathtt{tt} \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \\ &\longrightarrow\equiv \kappa[\![\mathbf{0}]\!] \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \mid \kappa \hookrightarrow_1 \overline{\kappa} : \mathtt{tt} \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \longrightarrow\equiv P' \end{aligned}$$

In this case, an exception and then `tt` are sent over κ. Finally the exception is delivered to $\overline{\kappa}$ before delivering `tt`. But we can also have:

$$\begin{aligned} P \longrightarrow\longrightarrow\equiv \mathbf{try}\{\ \mathbf{throw}\ \}\, \mathbf{catch}\, \{\, \kappa : \kappa!\langle \mathtt{tt}\rangle \,\} \mid \kappa \hookrightarrow_0 \overline{\kappa} : 5 \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \\ \longrightarrow\equiv \kappa[\![\kappa!\langle \mathtt{tt}\rangle]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : (\dagger :: 5) \mid \overline{\kappa}[\![\overline{\kappa}?(x)]\!] \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon \longrightarrow\longrightarrow\longrightarrow\equiv P' \end{aligned}$$

Above, 5 is sent over κ and an exception is thrown on $\overline{\kappa}$. In this situation, the system will ignore 5 (discarded by (WVAL)), and deliver `tt` inside the wrap.

The following example shows how refinement of an existing exception is handled:

$$R \;=\; \mathbf{try}\{\ \mathbf{try}\{\, \mathbf{throw} \,\}\, \mathbf{catch}\, \{\, (\kappa, \lambda) : Q_1 \,\}\ \}\, \mathbf{catch}\, \{\, \kappa : Q_2 \,\} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \mid \kappa \hookrightarrow_0 \overline{\kappa} : L$$

Process R either throws an exception in the inner try-catch block (by (THR)) or receives a remote exception (by (RTHR)). By applying (THR), (CLEAN) and (WTHR) in the first

case or by (RTHR) in the second case, we have (omitting some queues):

$$R \longrightarrow\equiv \mathbf{try}\{\ (\kappa,\lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L\ \}\ \mathbf{catch}\ \{\ \kappa : Q_2\ \} \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \longrightarrow$$
$$(\kappa,\lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L \mid \overline{\kappa} \hookrightarrow_0 \kappa : \dagger \longrightarrow (\kappa,\lambda)[\![Q_1]\!] \mid \kappa \hookrightarrow_0 \overline{\kappa} : \dagger :: L \mid \overline{\kappa} \hookrightarrow_1 \kappa : \epsilon$$

3 Typing Interactional Exceptions

This section introduces a type discipline for sessions with interactional exceptions. In comparison with the standard session types, the central difference is the shape of a type itself, which now consists of the abstraction of the default behaviour (the "try" part) and that of the handler behaviour (the "catch" part). This simple extension, combined with the use of levels, allows to establish subject reduction, guaranteeing that messages are always delivered at proper levels at proper timings in the presence of nested asynchronous escapes, testifying consistency of the operational semantics introduced in §2.

Type Syntax. The grammar of types extends the standard session types:

$$\begin{array}{rcl} \alpha,\beta & ::= & \downarrow(\theta).\,\alpha \mid \uparrow(\theta).\,\alpha \mid \oplus\{l_i : \alpha_i\}_{i\in I} \mid \&\{l_i : \alpha_i\}_{i\in I} \mid \alpha[\![\beta]\!] \mid \mathtt{end} \mid \mu\mathbf{t}.\,\alpha \mid \mathbf{t} \\ \theta & ::= & \langle \alpha[\![\beta]\!] \rangle \mid \mathtt{bool} \mid \ldots \end{array}$$

α and θ are respectively called *session types* and *service types*. The grammar follows the standard session types [10, 17], except for *try-catch type* $\alpha[\![\beta]\!]$, the abstraction of a try-catch block: in $\alpha[\![\beta]\!]$, α denotes the type of the try-block and β the catch block. A session type α is *plain* if it does not use a try-catch type (except in a service type it carries). From now on in $\alpha[\![\beta]\!]$, we stipulate α and β are both plain. This is because a try-catch on κ cannot occur nested in a try- or catch-block of λ if $\kappa = \lambda$.

The *dual* of α is written $\overline{\alpha}$. The dual of the try-catch type is defined as $\overline{\alpha[\![\beta]\!]} = \overline{\alpha}[\![\overline{\beta}]\!]$: the other cases are standard [10]. For example, by exchanging input and output, the dual of $\downarrow(\mathtt{string}).\mathtt{end}[\![\uparrow(\mathtt{bool}).\mathtt{end}]\!]$ is $\uparrow(\mathtt{string}).\mathtt{end}[\![\downarrow(\mathtt{bool}).\mathtt{end}]\!]$.

Environments. *Typing judgements* for processes and expressions have the forms $\Gamma \vdash P \triangleright \Delta$ 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

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

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 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 $\mathtt{bool}$. Typing $X : \Delta$ is used for recursion as in [6].

Typing System for Programs. We show the typing system by which the programmer can check whether her program is error free or not, especially w.r.t. its exception usage. The following are the selected typing rules:

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

$$(\text{TThr})\ \frac{\mathsf{fv}(\Gamma) = \emptyset}{\Gamma \vdash \mathbf{throw} \triangleright \prod_i \kappa_i :^0_u \alpha_i} \qquad (\text{TPar})\ \frac{\Gamma \vdash P_i \triangleright \Delta_i\ (i = 1, 2)\ \ \Delta_1 \asymp \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \odot \Delta_2}$$

Other than the rules for the exception constructs, all rules are identical to [21], augmented with annotation of exception levels.

(TReq) types a request on service channel c whose type, according to Γ, is $\alpha_j\{\!|\beta_j|\!\}$. Condition $s^+ = \kappa_j$ makes sure that the fresh name s^+ will also be in the try-catch after reduction. Session s^+ has type $\overline{\alpha}_j\{\!|\overline{\beta}_j|\!\}$, the dual of c's type. This rule checks that each κ_i in Q (exception handler) has type $\overline{\beta}_i$ whereas in P it has type $\overline{\alpha}_i\{\!|\overline{\beta}_i|\!\}$ where each $\overline{\beta}_i$ may come from a refinement of κ_i in P. Finally, Γ' is a subset of Γ without free variables for service channels (otherwise the queue stores open terms at run-time). In (TServ), because of SCP in § 2, services should never be prefixed therefore the only visible (free) session in P and Q should be s^-. Throwing an exception interrupts any conversation, thus (TThr) allows to type **throw** with any $\kappa : \alpha$ (unprotected). (TPar) requires the coherence relation $\asymp$ and the partial operator $\odot$ based on duality [10, 17]. When typing programs, the operator becomes just a set union. We shall extend it for types of queues in the next subsection. The rules for communication are standard.

Typing System for Run-Time Processes. The rules for typing run-time processes, which are necessary for type soundness, include, among others:

$$(\text{TExcept})\ \frac{\begin{array}{l}\Gamma \vdash P \triangleright \Delta \cdot \prod_j \lambda_j :^{n_j}_p \alpha'_j \cdot \prod_i \kappa_i :^{m_i}_\rho \alpha_i\{\!|\beta_i|\!\} \\ \Gamma' \vdash Q \rhd \prod_i \kappa_i :^0_u \beta_i \qquad \mathsf{queue}(\Delta) \qquad \Gamma' \subseteq \Gamma,\ \mathsf{fv}(\Gamma') = \emptyset\end{array}}{\Gamma \vdash \mathbf{try}\{\,P\,\}\,\mathbf{catch}\,\{\,\tilde{\kappa} : Q\,\} \triangleright \Delta \cdot \prod_j \lambda_j :^{n_j}_p \alpha'_j \cdot \prod_i \kappa_i :^{m_i}_p \alpha_i\{\!|\beta_i|\!\}}$$

$$(\text{TWrap})\ \frac{\Gamma \vdash Q \rhd \Delta \cdot \prod_i \lambda_i :^{n_i}_p \alpha'_i \cdot \prod_i \kappa_i :^0_u \beta_i \qquad \mathsf{queue}(\Delta)}{\Gamma \vdash \tilde{\kappa}\{\!|Q|\!\} \triangleright \Delta \cdot \prod_i \lambda_i :^{n_i}_p \alpha'_i \cdot \prod_i \kappa_i :^1_p \alpha_i\{\!|\beta_i|\!\}}$$

(TExcept) is a key rule, associating the type $\alpha\{\!|\beta|\!\}$ to the try-catch block, ensuring each κ_i is used as $\alpha_i\{\!|\beta_i|\!\}$ in P (β_i may come from a refinement in P) and as β_i in Q. The premise makes sure that each $\lambda_j \notin \tilde{\kappa}$ is protected in a try-catch block or a wrap in P. Without this condition, we may end up with unprotected code that could be brutally removed by an exception, violating the session duality. For example, in $\mathbf{try}\{\,\lambda!\langle v\rangle\mathbf{.}\ R\,\}\,\mathbf{catch}\,\{\,\kappa : Q\,\} \mid \overline{\lambda}?(x)\mathbf{.}\ R \mid P$, if an exception is thrown by P over κ, the output $\lambda!\langle v\rangle\mathbf{.}\ R$ would be lost leaving the input $\overline{\lambda}?(x)\mathbf{.}\ R$ alone violating duality. The predicate $\mathsf{queue}(\Delta)$ checks that Δ contains queue types only. No condition on m_i is required since in P, each κ_i can be either unprotected (no wraps nor try-catch blocks on κ_i) or protected (because of a refinement, κ_i occurs in a try-catch block or in a wrap). Finally, in order to record that now we have the try-catch block on κ_i, we force the tag

ρ to become p, i.e. protected. Note that, since Q is a program, we do not need to check that $\mathsf{fv}(\Gamma) = \emptyset$.

(TWRAP) types a wrap over a process Q. All the κ_i that have type β_i will have new type $\alpha_i\{\!|\beta_i|\!\}$ so to form the correct dual for the other side of the session in the case the exception has not been yet received there. The new type will make sure that each κ_i is protected. We set n to 1 in order to remember an existence of a wrap. As queues contain only an output or a select, queue types will only have output or selection types [12].

We conclude with the definition of $\asymp$ and $\odot$ using the treatment of queue types from [12]. We say Δ_1 and Δ_2 are *compatible*, written $\Delta_1 \asymp \Delta_2$, if and only if (i) $\mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2) = \emptyset$; (ii) $\kappa :^n_\rho \alpha$, $(\kappa, \overline{\kappa}) :\bot$, $\overline{\kappa} :^m_{\rho'} \beta$, $(\overline{\kappa}, \kappa) :\bot \in \Delta_1 \cup \Delta_2$ implies $\alpha = \overline{\beta}$; (iii) $\kappa :^n_\rho \alpha$, $(\kappa, \overline{\kappa}) : \alpha'$, $\overline{\kappa} :^m_{\rho'} \beta$, $(\overline{\kappa}, \kappa) : \beta' \in \Delta_1 \cup \Delta_2$ implies $\mathsf{merge}(\alpha', \alpha, n) = \overline{\mathsf{merge}(\beta', \beta, m)}$; and (iv) $\kappa :^n_\rho \alpha$, $(\kappa, \overline{\kappa}) :\bot$, $\overline{\kappa} :^m_{\rho'} \beta$, $(\overline{\kappa}, \kappa) : \beta' \in \Delta_1 \cup \Delta_2$ implies $\alpha = \overline{\mathsf{merge}(\beta', \beta, m)}$. The operation $\mathsf{merge}(\alpha', \alpha\{\!|\beta|\!\}, n)$ merges a session type with the type of the queue w.r.t. the level n i.e. it merges α' with α if $n = 0$ and it merges α' with β if $n = 1$. The operation $\Delta_1 \odot \Delta_2$ (defined if $\Delta_1 \asymp \Delta_2$) is such that if $\kappa :^n_\rho \beta$ and $(\kappa, \overline{\kappa}) : \alpha$ are in $\Delta_1 \cup \Delta_2$ then $\kappa :^n_\rho \mathsf{merge}(\alpha, \beta, n), (\kappa, \overline{\kappa}) :\bot \in \Delta_1 \odot \Delta_2$ ($\bot$ keeps track that the corresponding queue exists). The other elements in $\Delta_1 \odot \Delta_2$ are the same as in $\Delta_1 \cup \Delta_2$.

The processes in Examples 1/2 are typable: channel chSeller in both examples has type $\mu\mathbf{t}.\ \uparrow(\mathtt{int})\mathbf{.}\ \mathbf{t}\{\!|\downarrow(\mathtt{int})\mathbf{.}\ \uparrow(\mathtt{time})|\!\}$. In Example 2, channel chBroker has type $(\downarrow(\mathtt{int})\mathbf{.}\ \mu\mathbf{t}.\uparrow(\mathtt{int})\mathbf{.}\ \mathbf{t})\{\!|\oplus\{\mathtt{l}_1 :\downarrow(\mathtt{int})\mathbf{.}\ \uparrow(\mathtt{time}), \mathtt{l}_2 : \alpha\}|\!\}$ for some α.

In the following Theorem, $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

Theorem 1 (Subject Reduction). *Let P be a program such that $\Gamma \vdash P \triangleright \emptyset$. If $P \longrightarrow^* Q$ then $\Gamma \vdash Q \triangleright \emptyset$.*

As a corollary, the typing system also satisfies type safety and communication safety including communication-error freedom and linearity [12, Theorem 5.5].

4 Liveness

The previous section establishes a basic consistency of dynamics of typed processes via subject reduction. This section strengthens this result with a liveness property, which intuitively says that all compensating session channels eventually get fired (except those which are "erased" by thrown exceptions in which case the compensating actions both disappear). Along the way we also show relative termination of individual protocols for exceptions and their partial confluence vis-a-vis ordinary communications, two key consistency criteria for the proposed operational mechanism.

A Termination Protocol. We first augment the operational semantics in §2 with yet another protocol, called *termination protocol*, which in fact is an intrinsic part of the dynamics of exceptions. As an example, suppose there are two (and only two) processes in a configuration, which are try-catch blocks and which are communicating in a session. If each party's default process becomes the inaction process, it is natural to reduce each try-catch block to the inaction, freeing up the resources for its handler. This "garbage collection" is essential when we consider integration of interactional exception into the standard imperative programming languages with sequential composition since in this case launching Q depends on whether P reduces to the inaction or not.

We call a termination of a try-block in this form, *normal exit*. In interactional exceptions, a normal exit of a try-catch block demands an agreement of all peers: even if one try-block has terminated, if any of its communicating peers throws an exception, it also has to throw one, hence synchronising among all peers is essential for consistency. The termination protocol we introduce below makes the most of the tree structure associated with hierarchy of service invocation, leading to relatively efficient execution in terms of the number of messages. The protocol consists of two stages:

(**Stage 1: Voting)** Each try-catch block notifies its caller in the caller-callee relation of services by sending its vote of termination after itself terminating and receiving the same news from its callees.
(**Stage 2: Decision)** When the initial caller hears this, it in turn lets the news flow down to all of its callees, upon whose receipt the try-catch blocks can normally terminate.

Above, callees and callers refer to the service invocation mechanism. We formalise this protocol by extending the reduction relation. First, we augment polarities of channels in try-catch blocks with $\{\oplus, \ominus\}$, replacing $+, -$ for Stage 2 (indicating the casting of votes). We also use two special messages, $\{V, D\}$ standing for Voting and Decision.

$$
\begin{array}{ll}
(\textsc{Coll}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s'^{-}; \tilde{t}^{\oplus}; s^{+}; \tilde{r}^{+} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : V \quad \to \\
 & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s'^{-}; \tilde{t}^{\oplus}; s^{\oplus}; \tilde{r}^{+} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : \epsilon \\
(\textsc{Vote}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s^{-}; \tilde{t}^{\oplus} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : \epsilon \to \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s^{\ominus}; \tilde{t}^{\oplus} : \tilde{Q}\} \mid s^{-} \hookrightarrow_{\phi} s^{+} : V \\
(\textsc{Root}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{\tilde{s}^{\oplus} : \tilde{Q}\} \mid \textstyle\prod_i s_i^{+} \hookrightarrow_{\phi} s_i^{-} : \epsilon \to \star \mid \prod_i s_i^{+} \hookrightarrow_{\phi} s_i^{-} : D \\
(\textsc{Dec}) & \mathbf{try}\{\star\}\,\mathbf{catch}\,\{s^{\ominus}; \tilde{t}^{\oplus} : \tilde{Q}\} \mid s^{+} \hookrightarrow_{\phi} s^{-} : D \mid \textstyle\prod_i t_i^{+} \hookrightarrow_{\phi} t_i^{-} : \epsilon \quad \to \\
 & \star \mid s^{+} \hookrightarrow_{\phi} s^{-} : \epsilon \mid \textstyle\prod_i t_i^{+} \hookrightarrow_{\phi} t_i^{-} : D
\end{array}
$$

where $\star ::= \mathbf{0} \mid \star \mid \star \mid \tilde{\kappa}[\![\star]\!]$. Briefly, (Coll) collects the votes from the callees; (Vote) sends a vote to the caller once all the callees have voted; (Root) is for the initial caller which terminates once all its callees have voted; and (Dec) terminates once the caller has terminated. The rules only make sense for well-typed processes as we shall show later. We write $P \to_{term} P'$ for a reduction generated from the above rules. As an example, consider the following processes:

$$\mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{s^{-}; t^{+}; r^{+} : Q_1\} \mid \mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{s^{+} : Q_2\} \tag{1}$$

$$\mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{t^{-} : Q_3\} \mid \mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{r^{-} : Q_4\} \tag{2}$$

By (Vote) the two processes in (2) will put V in the queues with writing side r^{+} and t^{+} respectively. Applying (Coll) twice, the right-hand process in (1) will reach the state $\mathbf{try}\{\mathbf{0}\}\,\mathbf{catch}\,\{s^{-}; t^{\oplus}; r^{\oplus} : Q_2\}$. Again, by (Vote), it will send V to the parent. Finally, by (Root) and then (Dec) it will reduce to process $\mathbf{0}$.

Note that for the exit protocol to work, the caller-callee relation must have a tree structure where the root is the initial service invoker and leaves are a collection of interacting processes. We shall explore this topic further in the next subsection.

Normal and Exceptional Exits. We first show that the above protocol always leads to a normal exit. For this purpose, we need to identify the set of nodes that form the caller-callee relation in a given process.

Definition 2 (Node). *R is a* node process *of P whenever $P \equiv (\nu \tilde{s})\,(Q \mid R)$, R is restriction/queue/service-free, and R does not have the form $\star$ or $R' \mid R''$. If $\mathsf{fsc}(R) \neq \emptyset$ then $\mathsf{fsc}(R)$ is called* a node *of P.*

A node process of P is a subprocess of P which is not the parallel composition of two other processes and contains no restriction, queue, service or is composed by zero or more wraps over process $\mathbf{0}$. Then a node is the set of free session channels (only if non-empty) of a node process. Using the processes (1) and (2) given above, the process (1) | (2) has four nodes: s^+, (s^-, t^+, r^+), t^- and r^-. Process $\mathbf{try}\{\,\kappa!\langle v\rangle.\ \mathbf{0}\,\}\ \mathbf{catch}\ \{\,\kappa : Q\,\}$ has a unique node κ while $\mathbf{try}\{\,\kappa!\langle v\rangle.\ \mathbf{0}\,\}\ \mathbf{catch}\ \{\,\kappa : Q\,\} \mid \lambda?(x)$ also has λ. The caller-callee structure of a process is identified by the directed edges of the following graph.

Definition 3 (Invocation Graph). *Let G be the set of nodes of a process P. Then the* invocation graph of *P is the directed graph $\mathcal{G} = (G, E)$ where $(\tilde{\kappa}, \tilde{\lambda}) \in E$ if and only if $\kappa_i \in \{s^+, s^\oplus\}$ and $\lambda_j \in \{s^-, s^\ominus\}$ for some s, i, j. An* invocation tree *in P is a maximal subtree in the invocation graph.*

The invocation graph of process (1) | (2) is a graph with one edge from s^+ to (s^-, t^+, r^+), one from (s^-, t^+, r^+) to t^- and one from (s^-, t^+, r^+) to r^-. Recall the definition of programs given in § 2.1. If we reduce a typed program by zero or more steps then the invocation graph of the resulting process always forms a forest.

Lemma 4 (Evolution). *Let P be a typable program. If $P \to^* R$ then R's invocation graph $\mathcal{G}(R)$ is a forest.*

An invocation tree in P is in the *pretermination state* if each of its try-blocks have the form $\star$. From the Evolution Lemma, it follows that once an invocation tree reaches a pretermination state then all of its nodes will eventually vanish (see [2] for details).

Theorem 5 (Normal Exit). *Let P_0 be typable program and $P_0 \to^* P$ such that P has an invocation tree $\mathcal{T}$ in the pretermination state. Then whenever $P \to^* P'$ there is Q such that $P' \to^*_{term} Q$ where Q does not contain any active nodes from $\mathcal{T}$.*

The result above can actually be made stronger. For instance, in $\mathbf{try}\{\,\mathbf{try}\{\,\mathbf{0}\,\}\ \mathbf{catch}\ \{\,\lambda : Q_1\,\}\,\}\ \mathbf{catch}\ \{\,\kappa : Q_2\,\}$, we do not have a pretermination state as the outer try-block contains a try-catch block. Nevertheless, the normal exit is guaranteed as when the inner block and the subtree connected to λ terminate, the outer catch block can proceed.

A try-catch block can also exit due to an exception which will be propagated through the invocation graph. We write $P \to_{ex} P'$ if this is generated from (RTHR) and we say that κ is in *preexception* if it is the channel for a try-block which moreover contains an active **throw**.

Theorem 6 (Exception Exit). *Let P_0 be a typable program and $P_0 \to^* P$ such that P has an invocation tree $\mathcal{T}$. Suppose $\tilde{\kappa}$ is in a node of $\mathcal{T}$ which is in preexception for some κ_i. Then $P \to^* P'$ implies $P' \to^*_{ex} R$ for some R.*

Liveness. We can use the Evolution Lemma to obtain a strong form of liveness for well-typed processes in the presence of asynchronous exceptions. We first define:

Definition 7. *We say P is* stable *if P is the parallel composition of zero or more $\star$ processes and zero or more empty queues, possibly under ν-restrictions. We say P has* all resources *if $a \in \mathsf{fn}(P)$ implies $P \equiv (\nu\tilde{u})\ (*a(\lambda)[Q_1, Q_2] \mid R)$, i.e. all output channels are compensated by replicated inputs.*

We can finally state that a well-typed program either continues to reduce forever or reaches a state which does not contain active prefixes (except replicated services), try-catch blocks, throws nor messages in transit.

Theorem 8 (Liveness). *Suppose P is a typable program and has all resources. Then $P \to^* Q$ implies either Q is stable or $Q \to Q'$ for some Q'.*

In particular, if there are two compensating actions in a session at the same exception level, then these two will eventually interact, attesting the consistency of exception protocols. Practically, observing that SCP (the key reason the result holds) is widely found in e.g. services in the world wide web, the result says that if a conversation ever gets stuck in such an environment, it may as well be for non-interactional reasons (such as deadlock over shared resources at the servers).

5 Related Work and Conclusion

Related Work. In concurrent programming of distributed objects, exception handling is investigated in [20] where an algorithm to resolve multiple kinds of exceptions (which form a linear order) among concurrently running objects is proposed. Asynchronous exceptions among concurrent threads and their interplay with states in Haskell is studied in [16]. Motivated by subtle race conditions for mutual states, they formalise and implement blocking constructs to postpone asynchronous exceptions. The key idea is to relax the exception mask through the use of interruptable operations, to balance asynchrony and state consistency. [3] 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 [15] provides an operation to kill processes except those protected by wraps similar to our exception mechanism. CaSPiS [5], a session-based process calculus, is equipped with an operator for session closures. Our termination protocol, instead, is run whenever a try-block contains an inactive process, ensuring liveness. [18] introduces a calculus for web services by extending the π-calculus with service and request primitives and local exceptions, without asynchronous queues. An interesting idea is *context*, a named tree-like structure where a process is located. They do not have an explicit notion of session type. Their exceptions are the traditional local exceptions, without supporting propagation, coordinated transfer to a different part of a dialogue, nor the associated type abstraction, so that type checking protocols with exceptions, such as Examples 1/2 in §2, would be difficult.

The central focus of the present work is to have basic high-level typed abstractions for clear and flexible descriptions of conversation structures. Exceptions are asynchronously raised by multiple communicating peers, for which the session compatibility can guarantee type-safety in the presence of arbitrarily nested exceptions. These key aspects, backed up by safety and liveness properties relying on linearity of session-based communications, have not been investigated in the existing studies.

Further Results. For the sake of simplicity, we have restricted programs so that in $*c(\lambda)[P, Q]$ and $\overline{c}(\lambda)[\tilde{\kappa}, P, Q]$, the handler Q does not contain another try-catch at the same λ (try-catch is only used at run-time). An extension of our formalism allowing

try-catch to occur in the handlers of (service) and (request) would allow a process to "try" again after an exception has been thrown (cascading exceptions). For this purpose, try-catch types should be extended such that in $\alpha\{\!|\beta|\!\}$ the type α is always plain while β can be either plain or a try-catch type. Additionally, as it is now possible to have any number of nested wraps (when an exception is thrown several times), the number n in $\kappa :^{n}_{\rho} \alpha$ becomes arbitrary natural numbers, generalising their composition with queue types (n wraps). With essentially the same operational semantics, this generalised calculus satisfies the subject reduction and liveness properties. Further generalisation to existing session types is possible, including multiparty sessions [12] for flexible multicast exception propagation.

Further Topics. The key idea of the presented operational semantics is the use of exception levels in queues and their interplay with wrapped processes. In implementation, the queue level can be recorded in a header of each message which its receiver can check efficiently. The wrapping level can be part of a process state, recording its exception depth. Various optimisations are possible, for example dispensing with most coordination protocols when the handler type is trivial, obtaining essentially the same level of efficiency as local exception. In the near future, we plan to incorporate this exception mechanism to our on-going implementation of Java with session types [13].

For simplicity, we omit session delegations: we formulated this extension by storing frozen processes in queues. The type soundness holds by extending the typing rules with those in [10]. However a construction of the invocation graphs which can guarantee forest structures for the liveness property is left open.

Our liveness property, which involves the termination protocols, is similar to the property found in e.g. [7]. Apart from presence of exceptions, the aims and approaches of the two works are quite different: in the present work, the liveness is used for ensuring consistency of the proposed exception mechanisms, and the proof method is more operational, being applicable to any session-based calculus which has the service channel principle without delegation, without changing its typing system. On the other hand, in [7], an effect-based typing system is used for progress with delegation. It is an interesting further topic to incorporate our typing system with [7] for obtaining liveness with both exceptions and delegations.

A significant future topic is the treatment of multiple kinds of exception types in the present framework. Following [20], we may assume ordering on exception types (such as the exception class hierarchy as found in Java) for coordinating exceptions among multiple peers. Using a similar technique developed in our termination protocol, we can exploit the tree-structure of an invocation graph for efficient resolution of exception types to handle the subtle interplay between multiple exception types and nested exceptions for a more refined exception propagation.

With interactional exceptions, many practical scenarios can be accurately described through session primitives, and type-checked by our type theory. The syntax and type structures developed in this paper are being considered for use in a Web Services language (WS-CDL [6, 19]) and a language of message schemes for financial communications (ISO 20022 UNIFI [14]), throughout our collaborations with industry partners.

Acknowledgements. We thank the reviewers for their comments and our academic and industry colleagues for their stimulating conversations. This work is partially supported by EPSRC GR/T03208, EP/F002114 and EP/F003757, and IST2005-015905 MOBIUS.

References

1. Advanced Message Queuing Protocol. `http://www.iona.com/opensource/amqp/`.
2. Long version of this paper. `http://www.dcs.qmul.ac.uk/~carbonem/exception`.
3. L. Bocchi, C. Laneve, and G. Zavattaro. A calculus for long-running transactions. In *FMOODS'03*, volume 2884 of *LNCS*, pages 124–138. Springer, 2003.
4. E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC'07*, volume 4912 of *LNCS*, pages 240–256. Springer, 2008.
5. M. Boreale, R. Bruni, R. D. Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *FMOODS'08*, volume 5051 of *LNCS*, pages 19–38. Springer, 2008.
6. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
7. M. Dezani-Ciancaglini, U. de'Liguoro, and N. Yoshida. On Progress for Structured Communications. In *TGC'07*, volume 4912 of *LNCS*, pages 222–239. Springer, 2008.
8. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
9. S. Gay and V. T. Vasconcelos. Asynchronous functional session types. TR 2007–251, University of Glasgow, May 2007.
10. 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-Verlag, 1998.
11. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, 91:165–185, 2007.
12. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
13. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, LNCS. Springer, 2008. To appear.
14. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org/index.cfm?item_id=56664#interest`.
15. A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *ESOP '07*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
16. S. Marlow, S. L. P. Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in Haskell. In *PLDI*, pages 274–285. ACM, 2001.
17. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.
18. H. Vieira, L. Caires, and J. Seco. The conversation calculus: A model of service oriented computation. In *ESOP'08*, volume 4960 of *LNCS*, pages 269–283. Springer, 2008.
19. Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.
20. 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.
21. N. Yoshida and V. T. Vasconcelos. Language primitives and type disciplines for structured communication-based programming revisit. *ENTCS*, 171(4):73–93, 2007.