Math. Struct. in Comp. Science (2016), *vol.* 26, *pp.* 156–205. © Cambridge University Press 2014
doi:10.1017/S0960129514000164 First published online 10 November 2014

Global escape in multiparty sessions[†]

SARA CAPECCHI[‡], ELENA GIACHINO[§] and NOBUKO YOSHIDA[¶]

[‡]*Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy*
Email: sara.capecchi@unito.it
[§] *Dipartimento di Informatica, Università di Bologna/INRIA, Via Zamboni 33, 40126 Bologna, Italy*
Email: giachino@cs.unibo.it
[¶]*Department of Computing, Imperial College, South Kensington Campus, London SW7 2AZ, United Kingdom*
Email: n.yoshida@imperial.ac.uk

Received 31 January 2013; revised 1 October 2014

This article 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 realized 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 error handling for distributed cooperating threads. Guided by multiparty session types, our semantics is proven to provide a termination algorithm for global escapes. Our type system guarantees further safety and liveness properties, such as progress within the session and atomicity of escapes with respect to the subset of involved participants.

1. Introduction

In multiparty distributed conversations, a frequent communication pattern is the one providing 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* could represent either a computational error or a controlled, structured interruption requested by some participant. Despite the importance of error handling, there is a lack of support design and documentation for dedicated mechanisms. This is mainly due to the fact that the focus in development stage is on the 'normal' part of applications while error/escape detection and handling are left to implementation phase. In the context of distributed systems, this deficiency becomes even more serious; the handling of an escape of multiple cooperating peers from a specific point of the execution to another requires to reach a global agreement, to have a global view on the system state and to handle possible multiple concurrent errors (Castor Filho *et al.* 2009). In order to face the above aspects, error detection and handling mechanisms should be addressed at design level using formal methodologies (Rubira *et al.* 2005).

[†] This work has been partially supported by ICT COST Action IC1201 BETTY EPSRC EP/G015635/01, EP/K011715/01, EP/K034413, EP/L00058X/1 EU project FP7-231620 'HATS', EU project FP7-612985 UpScale, EU project ASCENS 257414, EU project FP7-610582 ENVISAGE MIUR Project 'CINA', Ateneo/CSP Project 'SALT' and by ANR 11 INSE 007 'REVER'.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

To this aim, in this article we propose a structured global escape mechanism based on multiparty session types, which can control multiple interruptions efficiently, and guarantees deadlock-freedom within the conversation. Session types have been widely studied as a type foundation for structured distributed, communication-centred programming, in the context of many process calculi and programming languages. The original binary session types (Honda *et al.* 1998; Takeuchi *et al.* 1994) have been generalized to multiparty (Honda *et al.* 2008) in order to enforce conformance to specified interaction structures among multiple cooperating participants. In multiparty session types, critical properties are guaranteed by the combination of a static type-checking methodology based on the existence of a global type (a description of a multiparty protocol from a global viewpoint) and of its end-point projections (the global type is projected to end-point types against which processes can be efficiently typed). Our main focus is to extend the multiparty asynchronous session model with *interactional exceptions* (Carbone *et al.* 2008), which perform not only local management of the interrupted flows but also explicitly coordinate a set of collaborating and communicating peers. This requires a consistent propagation of exception messages: in this case, 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. We consider asynchronous communications in order to model real-world examples. Interactional exceptions based on multiparty asynchronous sessions provide the following contributions:

— an extension of multiparty sessions (Honda *et al.* 2008) to flexible exception handling: we allow asynchronous escape at any desired point of a conversation, including nested exceptions;
— a flexible representation for modelling both 'light' exceptions, namely control flow mechanisms rather than errors (such as time-outs), and real error handling; the errors that can be treated in this model are those that do not destroy communication channels used for interaction;
— a compositional model where nested exceptions are inner isolated contexts involving only a subset of participants who can handle an unexpected situation without affecting the unrelated interactions among other cooperating peers;
— 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) (Rubira and Wu 1995).

Our extension is *consistent* (since despite asynchrony 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).

1.1. *Designs issues and choices*

In designing a model for exception handling in distributed interaction many important aspects have to be tackled:

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

i. How to deal with concurrent exceptions?
ii. How to notify all involved participants in a consistent way?
iii. How to structure exception scopes in order to enforce modularity?

To illustrate our model we present here a three-party use-case involving a client, a seller and a bank that models *criss-crossing global interaction* (Carbone *et al.* 2006). The interaction and the possible failures are summarized in Figure 1:

1. a client `C` sends an order to a seller `S`,
2. `S` sends `C` a message to accept the order,
3. `C` sends to the bank `B` the code of her account and then waits for `B`'s answer,
   3.1. if the answer is affirmative (`OK`) then
      3.1.1. `C` sends the money to `S`,
      3.1.2. `C` waits for the delivery date from `S`,
   3.2. if the answer from the bank is negative (`NEM`, not enough money) then `C` starts a negotiation with `B` for a loan:
      3.2.1. `B` sends iteratively loan-offers to `C` (offers are updated and sent to `C` at regular time intervals),
      3.2.2. in case of success `C` sends the money,
      3.2.3. `C` waits for the delivery date from `S`.

There are several points of the above interaction in which there can be global or local escapes, we just describe two examples:

— the client waits for the delivery date only for a limited amount of time then she can decide to abort;
— the client could decide to leave the negotiation with the bank because she accepts the loan. In this case, the client and the bank must check that the accepted loan is the last one (there could be a criss-cross messaging causing the client to accept the old offer).

The first case causes the transaction abortion thus involve all participants. The second one could involve only the bank and the client (in case the negotiation is concluded successfully) and should not affect the seller activities. This second case shows that there are escapes from interactions involving only a subset of participants and that can be handled without alerting the others. In order to enforce modularity we should treat these *sub*-interactions without affecting the whole system (or in general a wider sets of participants). If something goes wrong while handling the local escape, that is, the exception cannot be solved 'internally' anymore, the control can be passed to the handler of the enclosing interaction.

The three blocks in Figure 1 represent the parts of the interaction that could be involved in an escape: in the blocks with diagonal lines the client could decide to abort the transaction because she has waited too much for the seller confirmation; the grey block represent the escape from the client-bank negotiation. The interactions above can be implemented as try-catch blocks, a well-known construct in programming languages such as Java and C++. This constructs allow a thread of control in a block (often

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

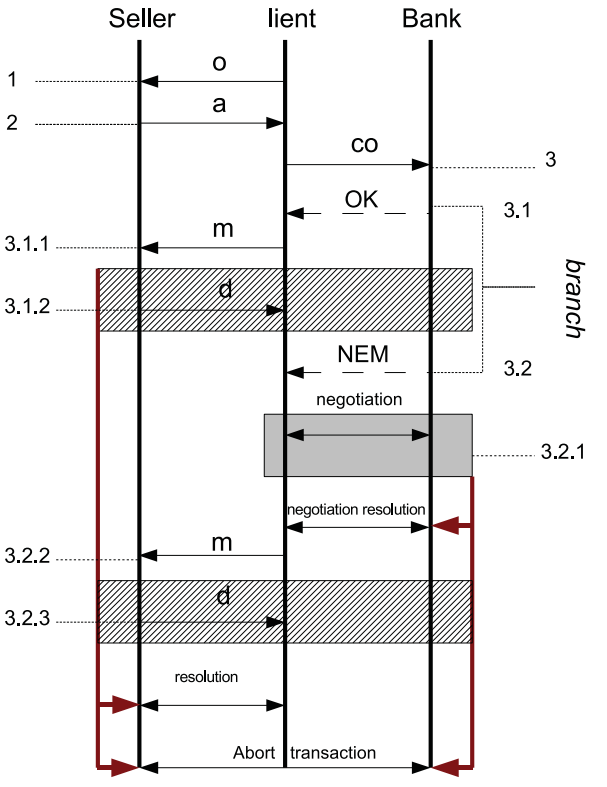


Fig. 1. The client-seller-bank example.

designated as *try block*) to move to another one (*exception handler, catch block*), when a system (or a user) raises an exception. They implement 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. For these reasons, they are the natural candidates to express exception contexts in our multiparty calculus together with the *throw* construct, used to raise an exception. Our try-catch blocks are of the shape:

$$\mathsf{try}(\tilde{s})\{P\}\ \mathsf{catch}\ \{Q\}$$

where:

— the try block has two arguments

- $\tilde{s}$ are the channels used for communications between participants in the interaction described by process P and Q.
- P represents the default interaction during which an exception could be thrown using *throw* construct.

— the argument of the *catch* block, Q, describes interactions during exception handling.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

For instance, in our example, the process implementing the execution on the client side can be represented as:

$$\mathsf{try}(s_1, s_2, s_3, s_4)\{P_1 ; \mathsf{try}(s_3, s_4)\{P_2\} \ \mathsf{catch} \ \{Q\} ; P_3\} \ \mathsf{catch} \ \{\mathtt{abort}\} \tag{1}$$

where:

— s_1, s_2, s_3, and s_4 are the channels used for communications.
— P_1 represents the interaction with the seller at points 1–2.
— $\mathsf{try}(s_3, s_4)\{P_2\} \ \mathsf{catch} \ \{Q\}$ represents the interaction with the bank at point 3. In particular, P_2 is the negotiation that can be interrupted and resolved within Q: if the client wants to accept the loan or to abandon the negotiation she can raise an exception moving to Q (this can be done by using in P_2 the construct $\mathsf{throw}(s_3, s_4)$ which indicates that an exception has been raised in the try block with channels s_3, s_4 as parameters). Because of the intrinsic asynchronous nature of exceptions, the client may have moved to Q accepting the loan, while the bank may have sent another offer before being notified of (and capturing) the exception (typical criss-cross scenario). Therefore, the client still has to check within Q that the accepted loan proposal is the last one offered. Thus in Q there are two different paths: (i) the negotiation is successful, in this case the execution of the try block terminates and the interaction goes on with P_3; (ii) the negotiation is not successful so the client throws an exception to abort the transaction by using the construct $\mathsf{throw}(s_1, s_2, s_3, s_4)$, which indicates that an exception has been raised in the try block with channels s_1, s_2, s_3, s_4 as parameters. This will move the interaction to the handler of the enclosing try block which will manage the abort of the transaction.
— P_3 represents the interaction with the seller at points 3.1.1, 3.1.2 or 3.2.2, that is the sending of the money by the client and the receiving of the date from the seller.

The implementation of the interaction from the seller(2) and the bank(3) point of views are:

$$\mathsf{try}(s_1, s_2, s_3, s_4)\{P_1' ; P_3'\} \ \mathsf{catch} \ \{\mathtt{abort}\} \tag{2}$$

$$\mathsf{try}(s_1, s_2, s_3, s_4)\{\mathsf{try}(s_3, s_4)\{P_2'\} \ \mathsf{catch} \ \{Q\}\} \ \mathsf{catch} \ \{\mathtt{abort}\} \tag{3}$$

where the communications in P_i', Q_i' are the complementary actions of those in P_i, Q_i of the client process (1) above, with $i \in \{1, 2, 3\}$.

In the processes (1) and (3) there are two try blocks: the first, which we call *external* (*enclosing, outer*), involves all participants and all channels; the second, which we call *internal* (*nested, inner*), involves only the client, the bank, and two channels, and confines the exception resolution between them.

We may represent the protocol of our example by the global type of Figure 2. The shape of the global type representing a try-catch block is $\{\tilde{k}, \gamma, \gamma'\}$ where $\tilde{k}$ is the set of involved channels represented by their index, γ is the global type of the default interaction and γ' is the global type of exception handling. Let us stress again that escapes are also used to implement a sort of 'go-to' mechanism (as in the protocol above where the client throws an exception to exit the negotiation since she has accepted the loan). On the other

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$
\begin{array}{lll}
1. & \{(1,2,3,4), & \mathtt{C} \to \mathtt{S} : 1\langle \mathsf{string} \rangle; \\
2. & & \mathtt{S} \to \mathtt{C} : 2\langle \mathsf{string} \rangle; \\
3. & & \mathtt{C} \to \mathtt{B} : 3\langle \mathsf{string} \rangle; \\
4. & & \{(3,4), \mathtt{B} \to \mathtt{C} : 4\{\mathtt{OK} : \epsilon,\ \mathtt{NEM} : \gamma,\ \gamma'\}; \\
5. & & \mathtt{C} \to \mathtt{S} : 1\langle \mathsf{money} \rangle; \\
6. & & \mathtt{S} \to \mathtt{C} : 2\langle \mathsf{date} \rangle, \\
7. & \epsilon\} &
\end{array}
$$

Fig. 2. The global type of the client-seller-bank example.

hand, the real meaning of an exception is that of an unexpected event that can occur in any point of the interaction. Thus, exceptions are not part of the global type since they are extemporaneous problematic situations and having those in the description of the protocol would require each implementation to have a throw in the same exact point.

The whole interaction is part of a try-catch block involving all channels (1, 2, 3, 4). The global type of the default interaction starts at line 1 and ends at line 8. If the default protocol fails the transaction is aborted, for this reason the global type of the handler in line 9 is ϵ which denotes the empty communication (line 9). Line 1 says that C sends a string (the order) to S using channel 1. Line 2 says that S sends a string (the confirmation) to C using channel 2. In line 3, C sends to B a string (bank account code) using channel 3, then in line 4 we find a nested try block involving channels 3 and 4. This nested block describes the interaction between the client and the bank. The global type of the default interaction says that B sends to C on channel 4 one of the labels OK or NEM. In case OK is sent there are no more communications and the interaction goes on from line 5. Instead, if NEM is sent, B and C negotiate following the global type γ while γ' describes the type of the handler solving the exceptions raised during negotiation. Lines 5–6 says that C sends money to S using channel 1 and S sends a date to C using channel 2.

Let us come back to our three initial questions:

i. *How to deal with concurrent exceptions?* Both the seller and the client (or the client and the bank) could raise an exception in the same try block: this situation can be managed in the handler processes, which will coordinate the resolution of the raised exceptions. Communications and exceptions raising and handling mechanisms are asynchronous: participants could raise an exception before being notified about another one (which always happen in real use cases). In our model, the handler processes can coordinate to manage multiple exceptions when every participant (i) has notified, or has been notified an exception and (ii) has moved to the handler.

ii. *How to notify all involved participants in a consistent way?* Due to the asynchronicity of our model we need to avoid that messages in a 'default' conversation get mixed up with those in an 'exception' conversation. This requirement drives the design of the semantics.

iii. *How to structure exceptions scopes in order to enforce modularity?* As pointed out in the above example there are escapes from interactions involving only a subset of participants that can be handled without alerting the others. Thus, we structure exception blocks by isolating nested ones. Nested try blocks involve only a subset of participants: when an exception is raised inside a nested try block only the participants

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

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

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

Fig. 3. Syntax.

involved in the block are alerted and moved to the handler processes. The other peers are not alerted unless an exception is explicitly raised with a throw on the external block. We assume that a subset of participants needs only a subset of channels for communications (this condition is enforced by the type system).

This article is a full and revised version of Capecchi *et al.* (2010) including a complete revision of the semantics, detailed definitions, updated related work and full proofs. It is organized as follows: in Section 2 we describe the syntax of our calculus. In Sections 3–5 we present semantics, typing and properties of our extension; Section 6 shows how a known distributed object model can be encoded with our calculus. Finally, Sections 7 and 8 close with related works and conclusions respectively.

2. Multiparty session processes with exceptions

We formalize the key ideas of global escapes in the context of a small process calculus based on the π-calculus with multiparty session primitives (Honda *et al.* 2008).

Informally, a session is a series of interactions and it is established among multiple parties via a service name, which represents a public interaction point. The participants must agree on the private channel names that will be used for the communication within the session.

We use s, z over channels, r over indexed channels (of the form $s^{(\varphi)}$), a over public service names, v over values, x over variables, X over process variables and l over labels. We adopt the notation $\tilde{s}$ as a shorthand for the set $s_1, \ldots, s_n$. Processes ranged over by P, Q and R and expressions ranged over by e, are described by the grammar in Figure 3.

Let us informally comment on the primitives of the language, whose operational semantics will be given in the next section.

The session connection is performed by the prefix process $\overline{a}[2..n](\tilde{s}).P$ (over the service name a, specifying the number $n-1$ of participants invited) by distributing a vector of freshly generated session channels $\tilde{s}$ to the remaining $n-1$ participants, each of shape $a[\mathtt{p}](\tilde{s}).P$, with $2 \leqslant \mathtt{p} \leqslant n$. All parties receive $\tilde{s}$, over which the actual session communications can now take place.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

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, with the construct $r \rhd \{l_i : P_i\}_{i\in I}$, so that it is possible for a partner to select a behaviour by sending on r the corresponding label, with $r \lhd l.P$. The try-catch construct $\mathsf{try}(\tilde{s})\{P\}\ \mathsf{catch}\ \{Q\}$ describes a process P (called *default process*) that communicates on the channels $\tilde{s}$. If some exception is thrown on $\tilde{s}$ before P has ended, the compensation handler Q is going to take over. The construct $\mathsf{throw}(\tilde{s})$ throws the exception on the channels $\tilde{s}$. Try-catch blocks can be nested in both default and handler processes:

— as pointed out in the introduction, inner try blocks in the default process P must involve a proper subset of the participants of the outer try block, and use only a subset of channels to communicate (this is enforced by the type system). The exception raised inside these inner blocks can be resolved by the directly 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 the outer set of channels.

— concerning the handler process, notice that it is considered at the same level as the try-catch block itself, thus the condition on smaller sets of channels is checked w.r.t. a more external try-catch block, if it exists. For instance, we allow

$$\mathsf{try}(\tilde{s})\{P\}\ \mathsf{catch}\ \{\mathsf{try}(\tilde{s}')\{P'\}\ \mathsf{catch}\ \{Q'\}\},$$

assuming that a possible outer try catch is on a set $\tilde{z}$ such that $\tilde{s} \subset \tilde{z}$ and $\tilde{s}' \subset \tilde{z}$.

We include in our calculus sequence $P\,;\,P$, by which synchronization behaviours such as joining and forking processes can be modelled (Baeten and Wejland 1990; Bettini *et al.* 2008). As in Honda *et al.* (2008), declarations include a set of variables $\tilde{x}$ and a set of fixed channels $\tilde{s}$. In recursion, process variables X_i would occur in $P_1 \dots P_n$ and in P zero or more times. In $X_i(\tilde{x}_i\tilde{s}_i)$ binder variables $\tilde{x}_i$ and channels $\tilde{s}_i$ should be pairwise different.

$fpv(P)$ and $fn(P)$, respectively denote the sets of free process variables and free service names in P. $dpv(\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I})$ denotes the set of process variables $\{X_i\}_{i\in I}$ introduced in $\{X_i(\tilde{x}_i\tilde{s}_i) = P_i\}_{i\in I}$.

As in Honda *et al.* (2008), 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. In order to guarantee communication consistency in our calculus queues are structured in levels: a queue s is parted in $s[0], s[1], \dots$. When performing an action on a channel s^{φ} (when φ is 0 we just write s), a process is going to write or read at the level φ of the queue associated to s (i.e. at $s[\varphi]$). However in the user written code the level is always zero ($\varphi = 0$), namely the programmer is not responsible of managing the queue levels, which are automatically set up at compile time (for more details, see Section 4). For a consistent semantics we require that a session connection, i.e. a session accept or request, can never occur in a try-catch block (this is enforced by the type system): indeed if an exception is captured and the handler is executed, the session inside the default process will disappear while having still some pending communications.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$\begin{array}{lcl}
\mathtt{S} & = & \overline{\mathtt{BS}}[2,3](s_1,s_2,s_3,s_4).\mathsf{try}(s_1,s_2,s_3,s_4)\{s_1?(o).s_2!\langle \mathtt{a}\rangle; s_1?(m).c = \mathtt{true}; s_2!\langle \mathtt{d}\rangle\} \\
& & \quad \mathsf{catch}\{\mathsf{try}(s_1,s_2,s_3,s_4)\{\mathsf{if}\ (c)\ \mathsf{then}\ \{s_2!\langle \mathtt{d}\rangle\}\mathsf{else}\ \mathsf{throw}(s_1,s_2,s_3,s_4)\} \\
& & \quad\quad \mathsf{catch}\{\mathtt{abort}\}\} \\
\\
\mathtt{C} & = & \mathtt{BS}[2](s_1,s_2,s_3,s_4). \\
& & \mathsf{try}(s_1,s_2,s_3,s_4)\{s_1!\langle \mathtt{o}\rangle; s_2?(a); P'; s_1!\langle \mathtt{m}\rangle; (s_2?(d) \mid \mathsf{throw}(s_1,s_2,s_3,s_4))\} \\
& & \quad \mathsf{catch}\{\mathsf{try}(s_1,s_2,s_3,s_4)\{s_2?(d)\}\mathsf{catch}\{\mathtt{abort}\}\} \\
\\
P' & = & s_3!\langle \mathtt{co}\rangle; \\
& & \mathsf{try}(s_3,s_4)\{s_4 \rhd \{\mathtt{OK} : \mathbf{0}, \mathtt{NEM} : \text{negotiation with the bank }\} \\
& & \mathsf{catch}\{\text{ solving or aborting negotiation }\}\} \\
\\
\mathtt{B} & = & \mathtt{BS}[3](s_1,s_2,s_3,s_4).\mathsf{try}(s_1,s_2,s_3,s_4)\{P''\}\mathsf{catch}\{\mathsf{try}(s_1,s_2,s_3,s_4)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\} \\
\\
P'' & = & s_3?(co). \quad \mathsf{try}(s_3,s_4)\{\mathsf{if}\ \mathit{enoughmoney}\ \mathsf{then}\ s_4 \lhd \mathtt{OK}.\mathbf{0} \\
& & \quad\quad \mathsf{else}\ s_4 \lhd \mathtt{NEM}.\ \text{negotiation with the bank} \\
& & \quad\quad \mathsf{catch}\{\text{ solving or aborting negotiation }\}
\end{array}$$

Fig. 4. Criss-cross example.

Note that most of examples with session types in the literature can be written under this condition.

Example 1 (criss-cross protocols). The syntax of our client-seller-bank example is in Figure 4, where we use different fonts for *variables* and `values` for the sake of readability. Let us consider default processes. `S` receives an order from a `C` on channel s_1, then sends back the acknowledgement on s_2 and she processes the order. `C` waits for the order acknowledgement then starts interaction P' where she sends to `B` its code account on s_3. `B` checks if there is enough money then, according to the result of the test, sends on s_4 `OK` or `NEM` (not enough money) to `C`. If the answer is `OK`, `C` sends the money to `S` on s_1 and `S` sends the delivery date to `C` on s_2. If the answer is `NEM`, `C` and `B` start to deal for a loan. If the negotiation is successful `C` sends the money to `S` on s_1 and waits for delivery date. In the above interactions there are two escape points: `C` could decide to abort the transaction because it has waited too much for the delivery date from `S` or maybe she could want to escape from the negotiation with `B`. This is achieved by putting the throw construct in these points. Thus, if `C` decides to abort the transaction because `S` is late, she throws an exception on all involved channels (that is on (s_1, s_2, s_3, s_4)) and the execution passes to the handlers.

The handler of `S` is again a try block on the same channels and checks whether the order has been completed: if it has not (i.e., $c =$ false) then the interaction is aborted by `S` by throwing an exception on (s_1, s_2, s_3, s_4); otherwise $c =$ true means that `C` has raised the exception just before receiving the delivery date from `S` on s_2; in this case, execution goes on normally since the code in the handler in this case is the same as the default processes. Thus $c =$ true corresponds to order completion: from this moment on the interaction must proceed even if `C` has decided differently (this is quite standard in business protocol specifications: when a client aborts too late the transaction, she is often compelled either to conclude the payment or to pay some penalty fees).

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

3. Operational semantics for multiparty exceptions

In this section, we extend the semantics of multiparty sessions with exceptions propagation and handling. For communication safety purposes, we assume a set of multilevel queues and indexed channels that are used by the processes: a message sent to a channel s^{φ} will be put on the φ-th level of the queue s.

Multilevel queues are used to avoid mismatches in message exchanges between default and handler processes. In a try-catch block $\mathsf{try}(\tilde{s})\{P\}\ \mathsf{catch}\ \{Q\}$ if P uses channels $s_1^{\varphi_1},\ldots,s_n^{\varphi_n}$ then Q uses $s_1^{\varphi_1+1},\ldots,s_n^{\varphi_n+1}$. In this way the communications inside the default processes use queues $s_1[\varphi_1],\ldots,s_n[\varphi_n]$ while, when an exception is raised, handlers processes use queues $s_1[\varphi_1+1],\ldots,s_n[\varphi_n+1]$.[†]

We assume that processes have all the channels already set to the correct index. For instance, consider $A \mid B \mid C$ where:

$$A = \mathsf{try}(s_1,s_2)\{P_1\}\ \mathsf{catch}\ \{Q_1\};\mathsf{try}(s_1)\{P_2\}\ \mathsf{catch}\ \{Q_2\}$$
$$B = \mathsf{try}(s_1,s_2)\{P_1'\}\ \mathsf{catch}\ \{Q_1'\} \qquad C = \mathsf{try}(s_1)\{P_2'\}\ \mathsf{catch}\ \{Q_2'\}$$

we expect processes P_1 and P_1' to use channels $s_1^{\varphi}, s_2^{\varphi}$ (φ may be 0, if this is a top-level conversation), processes Q_1 and Q_1' to use channels $s_1^{\varphi+1}, s_2^{\varphi+1}$, processes P_2 and P_2' to use channel s_1^{φ}, processes Q_2 and Q_2' to use channel $s_1^{\varphi+1}$. The reuse of levels φ and $\varphi+1$ is safe because of the garbage collection by the reduction rule (RThr) (see detailed explanation of this rule). As we will see in Section 4, a well-formed global type describes the communication protocol between multiple participants (for instance A, B and C above) using channels with the appropriate levels so that the numbering of channels may be performed automatically by a preprocessing function inspecting the local process and the global type.

Reduction rules are defined in Figures 5 and 6. The reduction system uses an *exception environment* Σ, which keeps track of the raised exceptions. Σ will be used in rules (Thr), (RThr) and (ZThr). We also need *evaluation contexts* defined by the following grammars:

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

with the usual semantics: if a process P reduces to P', then $\mathcal{E}[P]$ reduces to $\mathcal{E}[P']$. The context $\mathcal{E}$ is used for structural reduction, in a standard way, in order to specify that it is possible to reduce inside a recursive definition, inside the first element of a sequential composition, inside any element in a parallel composition, within a restriction, and in the default process of a try-catch block. In rule (Link), a connection can be established only if all request/accept operations are at top level (it cannot happen that one of them is inside a try-catch block, or a restriction). In order to express that we need to specify a separate context $\mathcal{C}$ defined only for recursive definitions and sequential composition. Notice that the whole composition of processes trying to establish a session can, on the contrary, be

[†] In practice, multilevel queues can be implemented by recording the level in messages, together with the value (e.g. a pair of a value and a level), rather than creating n distinct queues per channel. Note that in this case neither the programmer has to deal with the level numbering explicitly.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$\text{(Link)}\quad \Sigma \vdash \mathcal{C}_1[\overline{a}[2..n](\tilde{s}).P_1] \mid \mathcal{C}_2[a[2](\tilde{s}).P_2] \mid \ldots \mid \mathcal{C}_n[a[n](\tilde{s}).P_n] \longrightarrow \Sigma \vdash (\nu \tilde{s})\,(\mathcal{C}_1[P_1] \mid \mathcal{C}_2[P_2] \mid \ldots \mid \mathcal{C}_n[P_n] \mid s_1[0] : \varnothing \mid \ldots \mid s_m[0] : \varnothing)$$

$$\text{(Send1)}\quad \frac{\tilde{e} \downarrow \tilde{v}}{\Sigma \vdash \mathcal{E}[s^{\varphi}!\langle \tilde{e} \rangle] \mid s[\varphi] : L \longrightarrow \Sigma \vdash \mathcal{E} \mid s[\varphi] : (L :: \tilde{v})} \qquad \text{(Send2)}\quad \frac{\tilde{e} \downarrow \tilde{v} \quad s[\varphi]\ \mathit{undefined}}{\Sigma \vdash \mathcal{E}[s^{\varphi}!\langle \tilde{e} \rangle] \longrightarrow \Sigma \vdash \mathcal{E} \mid s[\varphi] : \tilde{v}}$$

$$\text{(Sel1)}\quad \Sigma \vdash \mathcal{E}[s^{\varphi} \lhd l.P] \mid s[\varphi] : L \longrightarrow \Sigma \vdash \mathcal{E}[P] \mid s[\varphi] : (L :: l)$$

$$\text{(Sel2)}\quad \frac{s[\varphi]\ \mathit{undefined}}{\Sigma \vdash \mathcal{E}[s^{\varphi} \lhd l.P] \longrightarrow \Sigma \vdash \mathcal{E}[P] \mid s[\varphi] : l}$$

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

$$\text{(Branch)}\quad \Sigma \vdash \mathcal{E}[s^{\varphi} \rhd \{l_i : P_i\}_{i \in I}] \mid s[\varphi] : (l_j :: L) \longrightarrow \Sigma \vdash \mathcal{E}[P_j] \mid s[\varphi] : L \quad (j \in I)$$

$$\text{(If-T)}\quad \frac{e \downarrow \mathsf{true}}{\Sigma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow P} \qquad \text{(If-F)}\quad \frac{e \downarrow \mathsf{false}}{\Sigma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \longrightarrow Q}$$

$$\text{(Def)}\quad \frac{(\tilde{e} \downarrow \tilde{v}, X(\tilde{x}\tilde{s}) = P \in D)}{\Sigma \vdash \mathsf{def}\ D\ \mathsf{in}\ (X\langle \tilde{e}\tilde{s} \rangle \mid Q) \longrightarrow \Sigma \vdash \mathsf{def}\ D\ \mathsf{in}\ (P\{\tilde{v}/\tilde{x}\} \mid Q)}$$

$$\text{(Eval)}\quad \frac{\Sigma \vdash P \longrightarrow \Sigma' \vdash P'}{\Sigma \vdash \mathcal{E}[P] \longrightarrow \Sigma' \vdash \mathcal{E}[P']} \qquad \text{(Str)}\quad \frac{P \equiv P' \quad \Sigma \vdash P' \longrightarrow \Sigma \vdash Q' \quad Q \equiv Q'}{\Sigma \vdash P \longrightarrow \Sigma \vdash Q}$$

Fig. 5. Standard reduction rules.

$$\text{(Thr)}\quad \frac{\tilde{s}^{\tilde{\psi}} \geq \tilde{s}^{\tilde{\varphi}} \quad \mathit{implies} \quad \mathsf{throw}(\tilde{s}^{\tilde{\psi}}, \tilde{r}) \notin \Sigma}{\Sigma \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{\mathcal{E}[\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})]\}\ \mathsf{catch}\ \{Q\} \longrightarrow \Sigma \cup \mathsf{throw}(\tilde{s}^{\tilde{\varphi}}) \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{\mathcal{E}\}\ \mathsf{catch}\ \{Q\}}$$

$$\text{(RThr)}\quad \frac{\tilde{s}^{\tilde{\psi}} \geq \tilde{s}^{\tilde{\varphi}}\ \mathit{and}\ \left(\tilde{z}^{\tilde{\varphi}'} \subset \tilde{s}^{\tilde{\varphi}}\ \mathit{and}\ \mathsf{throw}(\tilde{z}^{\tilde{\psi}'}) \in \Sigma\ \mathit{and}\ \tilde{z}^{\tilde{\psi}'} \geq \tilde{z}^{\tilde{\varphi}}\right) \quad \mathit{implies} \quad \mathsf{try}(\tilde{z}^{\tilde{\psi}'})\ldots \notin P}{\Sigma, \mathsf{throw}(\tilde{s}^{\tilde{\psi}}) \vdash \quad \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{P\}\ \mathsf{catch}\ \{Q\} \mid (\textstyle\prod_i s_i[\varphi_i] : L_i)_{\{s_i^{\varphi_i} \in \mathsf{ch}(P)\}} \longrightarrow \Sigma, \mathsf{throw}(\tilde{s}^{\tilde{\psi}}) \vdash Q \mid (\textstyle\prod_i s_i[\varphi_i] : \varnothing)_{\{s_i^{\varphi_i} \in \mathsf{ch}(P)\}}}$$

$$\text{(ZThr)}\quad \frac{\tilde{s}^{\tilde{\psi}} \geq \tilde{s}^{\tilde{\varphi}} \quad \mathit{implies} \quad \mathsf{throw}(\tilde{s}^{\tilde{\psi}}) \notin \Sigma \qquad \Sigma' = \Sigma \setminus \{\mathsf{throw}(\tilde{z}^{\tilde{\varphi}'}) | \tilde{z} \subseteq \tilde{s}\}}{\Sigma \vdash (\nu \tilde{s}')(\textstyle\prod_i \mathcal{E}_i[\mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{\mathbf{0}\}\ \mathsf{catch}\ \{Q_i\}])_{i \in 1..n} \longrightarrow \Sigma' \vdash (\nu \tilde{s}')(\textstyle\prod_i \mathcal{E}_i)_{i \in 1..n}}$$

Fig. 6. Reduction rules for exception handling.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$P \equiv Q \quad \text{if } P =_\alpha Q$$
$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$
$$(\nu n)P \mid Q \equiv (\nu n)(P \mid Q) \quad \text{if } n \notin fn(Q)$$
$$(\nu nn')P \equiv (\nu n'n)P \qquad (\nu n)\mathbf{0} \equiv \mathbf{0} \qquad \mathsf{def}\ D\ \mathsf{in}\ \mathbf{0} \equiv \mathbf{0}$$
$$\mathsf{def}\ D\ \mathsf{in}\ (\nu n)P \equiv (\nu n)\mathsf{def}\ D\ \mathsf{in}\ P \quad \text{if } n \notin fn(D)$$
$$(\mathsf{def}\ D\ \mathsf{in}\ P) \mid Q \equiv \mathsf{def}\ D\ \mathsf{in}\ (P \mid Q) \quad \text{if } dpv(D) \cap fpv(Q) = \emptyset$$
$$\mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ D'\ \mathsf{in}\ P) \equiv \mathsf{def}\ D\ \mathsf{and}\ D'\ \mathsf{in}\ P \quad \text{if } dpv(D) \cap dpv(D') = \emptyset$$
$$\mathsf{try}(\tilde{s})\{(\nu n)P\}\ \mathsf{catch}\ \{Q\} \equiv (\nu n)\ \mathsf{try}(\tilde{s})\{P\}\ \mathsf{catch}\ \{Q\} \quad \text{if } n \notin fn(Q)$$

Fig. 7. Structural congruence.

in such a nested position since the problem arises only when some of the inviter/invitees are nested.

3.1. *Standard reduction rules*

Rule (LINK) establishes the connection among n peers on the private channels $\tilde{s}$, which are used for the communications within the session. One queue, with level 0, for each private channel is produced. Rules (SEND1,2) and (SEL1,2) push values and labels respectively into the queue $s[\varphi]$ corresponding to channel s^{φ}. The difference is that rule (SEND1) and (SEL1) are applied if $s[\varphi]$ has already been initialized, instead, rule (SEND2), (SEL2) take care of the initialization.

Rules (RECV), (BRANCH) perform the complementary input operations.

Let us notice that (as in Honda *et al.* (2008)) in rule (DEF), $\tilde{s}$ is a set of fixed channels thus only variables $\tilde{x}$ are substituted. Rules for conditionals and recursive definitions are standard. As usual, we consider processes modulo structural congruence (Figure 7).

3.2. *Reduction rules for exception handling and propagation*

The most delicate point that has to be managed is the handling of concurrent exceptions. Concurrent exceptions could be raised in both an internal try block and an external enclosing one, or by two peers in the same try block.

Even if two exceptions are raised concurrently their notification, that is the updating of Σ, is sequential. For instance consider the following process $P \mid P'$:

$$\Sigma \vdash \overbrace{\underbrace{\mathsf{try}(s_1,s_2)\{\overbrace{\mathsf{try}(s_1)\{\mathsf{throw}(s_1) \mid P_1\}\ \mathsf{catch}\ \{Q_1\}}^{P''}\}\ \mathsf{catch}\ \{Q\}}_{P} \mid \underbrace{\mathsf{try}(s_1,s_2)\{\mathsf{throw}(s_1,s_2) \mid \mathsf{try}(s_1)\{P_1'\}\ \mathsf{catch}\ \{Q_1'\}\}\ \mathsf{catch}\ \{Q'\}}_{P'}}$$

If both $\mathsf{throw}(s_1)$ and $\mathsf{throw}(s_1,s_2)$ are raised concurrently there are two cases:

— the external exception $\mathsf{throw}(s_1,s_2)$ is notified (that is, it is put into Σ) first. Then:

1. P' starts the execution of its corresponding handler Q'.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

2. P should notify the internal exception $\mathsf{throw}(s_1)$ but it finds the notification of the external exception $\mathsf{throw}(s_1, s_2)$ in Σ.

— the internal exception $\mathsf{throw}(s_1)$ is notified first. Then:

1. P'' starts the execution of its corresponding handler Q_1.
2. P' should notify the external exception $\mathsf{throw}(s_1, s_2)$ but it finds the notification of the internal exception $\mathsf{throw}(s_1)$ in Σ.

Concerning the scenarios above the semantics of exception handling and propagation must be designed taking into account the following key-points:

— when an exception is raised, the default process (including internal try-catch blocks) is aborted and the execution moves to the handler. Thus, in case the external exception is notified first, the inner try block is going to be aborted: there is no use in notifying and then starting to handle the inner exception. In this case, the outer exception has the priority;
— in case the inner exception is notified first, some process (P'' in the example) could move to the execution of the inner handler (Q_1 in the example) before catching the external exception. In this case all involved peers must cooperate, handling the inner exception as well, before catching the outer one. In this case, the inner exception has the priority.

Now let us explain the corresponding reduction rules in details.

Below we write $\tilde{s}^{\tilde{\psi}} \geqslant \tilde{s}^{\tilde{\varphi}}$ meaning that $\forall s_i : s_i^{\psi_i} \in \tilde{s}^{\tilde{\psi}}$ and $s_i^{\varphi_i} \in \tilde{s}^{\tilde{\varphi}}$ we have $\psi_i \geqslant \varphi_i$.

Rule (Thr) applies when the exception is thrown locally and has to be notified to all the participants involved in the current try block. In this rule a $\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})$ can be added to the environment in order to acknowledge all the try-catch blocks on the same set of channels $\tilde{s}$. As explained above this environment update is performed unless some exception has been thrown in an embedding block, that is, if Σ contains a throw on set of channels including the current set $\tilde{s}$. If Σ contains $\mathsf{throw}(\tilde{s}^{\tilde{\psi}} \cup \tilde{r})$, there are two cases :

1. $\mathsf{throw}(\tilde{s}^{\tilde{\psi}} \cup \tilde{r})$ has been raised in an outer try-catch block embedding the current one. This means that the current block is going to disappear since it is enclosed in the default process of an outer block, that will be discarded when the exception is captured. Thus, in this case Σ is not updated with $\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})$.
2. $\mathsf{throw}(\tilde{s}^{\tilde{\psi}} \cup \tilde{r})$ has been raised in an 'old' try-catch block, namely it is related to an already captured exception. In this case Σ is updated with $\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})$.

Case 1 holds if $\tilde{s}^{\tilde{\psi}} \geqslant \tilde{s}^{\tilde{\varphi}}$. Case 2 holds if there is at least a channel in the throw with an index lower w.r.t. the same channel in the current block ($\exists s_i : s_i^{\psi_i} \in \tilde{s}^{\tilde{\psi}}$, $s_i^{\varphi_i} \in \tilde{s}^{\tilde{\varphi}}$ and $\varphi_i > \psi_i$). For an explanatory example see Example 2.

Rule (RThr) reduces an exception $\mathsf{throw}(\tilde{s}^{\tilde{\psi}})$ that has been thrown and has been put in Σ. Let us consider the conditions in the premises:

— $\tilde{s}^{\tilde{\psi}} \geqslant \tilde{s}^{\tilde{\varphi}}$ checks that $\mathsf{throw}(\tilde{s}^{\tilde{\psi}})$ is not an old throw.
— $\left(\tilde{z}^{\tilde{\varphi}'} \subset \tilde{s}^{\tilde{\varphi}} \text{ and } \mathsf{throw}(\tilde{z}^{\tilde{\psi}'}) \in \Sigma \text{ and } \tilde{z}^{\tilde{\psi}'} \geqslant \tilde{z}^{\tilde{\varphi}'}\right)$ implies $\mathsf{try}(\tilde{z}^{\tilde{\psi}'}) \ldots \notin P$ checks that no inner exception has been previously notified: this would mean that another peer could be executing the internal handler. In order to be consistent with it, the current

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

process must catch the same internal exception before catching the external one. Again the condition on queue levels checks that $\mathsf{throw}(\tilde{z}^{\tilde{\psi}'})$ is not old.

If, according to premises, the throw can be reduced the execution passes to the handler Q and queues used by P are cleaned $((\prod_i s_i[\varphi_i] : \varnothing)_{\{s_i^{\varphi_i} \in \mathsf{ch}(P)\}})$.

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 all the handlers have to interact. Hence, we consider all the try-catch blocks and when every peer has terminated then they all can go on and the Σ can be cleaned by eliminating old throws: Σ' is obtained by eliminating from Σ all the throws raised on the current set of channels $\tilde{s}$ and its subsets.

Example 2. In this example, we show how our reduction rules deal with exceptions. As we have already said, we assume queue levels to be set up in a proper way by a preprocessing function (see Section 4.2.3 for details). Therefore, in this example we see that in the handler of a try-catch block on channel s_1 there is a $\mathsf{throw}(s_1^1, s_2)$. This is because the channel s_1 has been upgraded, being the parameter of the corresponding try, while s_2 is still in the scope of the default process that has s_2 among its try-parameters, thus it has not been upgraded.

Let us consider the reduction of the following process:

$$\varnothing \vdash \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{\mathsf{throw}(s_1) \mid P_1\} \ \mathsf{catch}\ \{\mathsf{throw}(s_1^1, s_2) \mid Q_1\}\} \ \mathsf{catch}\ \{Q\} \mid \mathsf{try}(s_1, s_2)\{\mathsf{throw}(s_1, s_2) \mid \mathsf{try}(s_1)\{P_1'\} \ \mathsf{catch}\ \{Q_1'\}\} \ \mathsf{catch}\ \{Q'\}.$$

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_1^1, s_2): we model a situation in which if the exception handling in the enclosed block fails, the outer one is alerted to handle the failure. The default process in the second line contains a throw on (s_1, s_2). Now let us analyse two possible scenarios.

Case 1. Let us suppose that $\mathsf{throw}(s_1)$ is raised first, by applying rule (Thr):

$$\mathsf{throw}(s_1) \vdash \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{P_1\} \ \mathsf{catch}\ \{\mathsf{throw}(s_1^1, s_2) \mid Q_1\}\} \ \mathsf{catch}\ \{Q\} \mid \mathsf{try}(s_1, s_2)\{\mathsf{throw}(s_1, s_2) \mid \mathsf{try}(s_1)\{P_1'\} \ \mathsf{catch}\ \{Q_1'\}\} \ \mathsf{catch}\ \{Q'\}.$$

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

$$\begin{array}{lll} \mathsf{throw}(s_1) & \vdash & \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{P_1\} \ \mathsf{catch}\ \{\mathsf{throw}(s_1^1, s_2) \mid Q_1\}\} \ \mathsf{catch}\ \{Q\} \mid \\ \mathsf{throw}(s_1, s_2) & & \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{P_1'\} \ \mathsf{catch}\ \{Q_1'\}\} \ \mathsf{catch}\ \{Q'\}. \end{array}$$

Then, by rule (RThr) we handle the one corresponding $\mathsf{throw}(s_1)$ (remember that the rule premise forbids us to reduce the external exception):

$$\begin{array}{lll} \mathsf{throw}(s_1) & \vdash & \mathsf{try}(s_1, s_2)\{\mathsf{throw}(s_1^1, s_2) \mid Q_1\} \ \mathsf{catch}\ \{Q\} \mid \\ \mathsf{throw}(s_1, s_2) & & \mathsf{try}(s_1, s_2)\{Q_1'\} \ \mathsf{catch}\ \{Q'\}. \end{array}$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Now we can apply (i) either rule (RTHR) to reduce $\mathsf{throw}(s_1, s_2)$ in Σ or (ii) rule (THR) to evaluate $\mathsf{throw}(s_1^1, s_2)$. In the last case, Σ has to be updated adding $\mathsf{throw}(s_1^1, s_2)$ and we have:

$$\begin{array}{ll} \mathsf{throw}(s_1) & \vdash \mathsf{try}(s_1, s_2)\{\mathsf{throw}(s_1^1, s_2) \mid Q_1\} \ \mathsf{catch}\ \{Q\} \mid \\ \mathsf{throw}(s_1, s_2), \mathsf{throw}(s_1^1, s_2) & \quad \mathsf{try}(s_1, s_2)\{Q_1'\} \ \mathsf{catch}\ \{Q'\}. \end{array}$$

In both cases, we can apply rule (RTHR) and the try blocks on s_1, s_2 can reduce to the handlers with the only difference that in case (ii) Σ contains also $\mathsf{throw}(s_1^1, s_2)$:

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

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

$$\begin{array}{rl} \mathsf{throw}(s_1, s_2) \vdash & \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{P_1\} \ \mathsf{catch}\ \{\mathsf{throw}(s_1, s_2) \mid Q_1\}\} \ \mathsf{catch}\ \{Q\} \mid \\ & \mathsf{try}(s_1, s_2)\{\mathsf{try}(s_1)\{P_1'\} \ \mathsf{catch}\ \{Q_1'\}\} \ \mathsf{catch}\ \{Q'\}. \end{array}$$

We apply rule (THR) to the inner $\mathsf{throw}(s_1)$ which is not added to the environment since an enclosing throw is already present in Σ. Then rule (RTHR) is applied twice to obtain:

$$\mathsf{throw}(s_1, s_2) \vdash Q \mid Q'.$$

Now let us assume $Q = \mathsf{try}(s_1^1, s_2^1)\{R_1\} \ \mathsf{catch}\ \{R_2\}$ and $Q' = \mathsf{try}(s_1^1, s_2^1)\{R_3\} \ \mathsf{catch}\ \{R_4\}$:

$$\mathsf{throw}(s_1, s_2) \vdash \mathsf{try}(s_1^1, s_2^1)\{R_1\} \ \mathsf{catch}\ \{R_2\} \mid \mathsf{try}(s_1^1, s_2^1)\{R_3\} \ \mathsf{catch}\ \{R_4\}\ .$$

By applying semantics rule the old $\mathsf{throw}(s_1, s_2)$ is ignored because the levels are all smaller than the one in the try block.

For further examples, see Appendix A and B. We just mention here that thanks to our semantics the implementation of our client-bank-seller protocol never moves to the situation where:

— the seller sends a confirmation to the client but the client aborts the interaction.
— the client accepts the wrong loan offer from the bank.

4. Typing structured global escapes

This section extends the definition of global types and the type system in Honda *et al.* (2008) to exception handling constructs. Given a global protocol G, we first apply a function *Qlevel* in order to annotate the channels with the right levels (Definition 4.1), then a projection function produces a local specification for each participant (Section 4.3), finally each participant's code is annotated and type-checked w.r.t. the local specification (Section 4.4). If the whole procedure succeeds, then the participants can communicate in a safe way.

In particular, our extended typing enforces some design choices related to exception handling:

i. the enclosed try-catch block must be listening on a smaller set of channels. We need 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. This is ensured by the well formedness of the global type (see Section 4.2);

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

ii. no session request or accept can occur inside a try-catch block as we explained in Section 2. This is enforced by the type system (see Section 4.4);

iii. after each exception is encountered the communication is moved on a new level of the queue, in a consistent way, as explained in Section 3. This is enforced by checking well formedness of global types (see Section 4.2).

4.1. *Types*

The syntax of types distinguishes 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:

$$\begin{array}{lll} \textit{Partial} & \gamma & ::= \mathtt{p}_1 \to \mathtt{p}_2 : k\langle \tilde{S} \rangle \mid \mathtt{p}_1 \to \mathtt{p}_2 : k\{l_i : \gamma_i\}_{i \in I} \mid \\ & & \{\!|\tilde{k}, \gamma, \gamma|\!\} \mid \gamma ; \gamma \mid \gamma \parallel \gamma \mid \mu \mathbf{t}.\gamma \mid \mathbf{t} \mid \epsilon \\ \textit{Global} & G & ::= \gamma ; \mathsf{end} \qquad \textit{Sorts} \quad S ::= \mathsf{bool} \mid \ldots \mid \langle G \rangle \end{array}$$

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. Type $\gamma \parallel \gamma'$ represents concurrent run of communications described by γ and γ'. Type $\mu \mathbf{t}.\gamma$ is a recursive type for recurring interaction structures, assuming type variables $\mathbf{t}$ are guarded in the standard way, i.e. type variables only appear under the prefixes (hence contractive). We take an equi-recursive view, not distinguishing between $\mu \mathbf{t}.\gamma$ and its unfolding $\gamma[\mu \mathbf{t}.\gamma/\mathbf{t}]$. We assume that $\langle G \rangle$ in the grammar of sorts is closed, i.e. without type variables. Type end represents the termination of the session and is often omitted. We identify $G \parallel \mathsf{end}$ and $\mathsf{end} \parallel G$ with G.

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

$$\begin{array}{lll} \textit{Partial} & \alpha, \beta & ::= k!\langle \tilde{S} \rangle \mid k?(\tilde{S}) \mid k \oplus \{l_i : \alpha_i\}_{i \in I} \mid k\&\{l_i : \alpha_i\}_{i \in I} \mid \\ & & \{\!|\tilde{k}, \alpha, \alpha|\!\} \mid \mu \mathbf{t}.\alpha \mid \mathbf{t} \mid \epsilon \mid \alpha ; \alpha \\ \textit{Action} & A, B & ::= \alpha \mid \alpha ; \mathsf{end} \mid \mathsf{end} \end{array}$$

As in Honda *et al.* (2008), 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 $k!\langle \tilde{S} \rangle$ 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 s_k then they behave as α_i. The remaining types are a local version of the global ones.

Example 3. We recall that the global type related to the client-seller-bank example has been described in Section 1 (Figure 2). The projection of that global type on the client

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

side is:

$$
\begin{array}{l}
1!(\mathsf{string});\\
\{(1,2,3,4), 3!(\mathsf{string}\{(3,4), 4 \oplus \{\mathtt{OK} : \epsilon, \mathtt{NEM}\ \mu\mathbf{t}.4?(\mathsf{double});\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 3\&\{\mathtt{OK} : 1?(\mathsf{float}), \mathtt{NOK} : \mathbf{t}\};\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 4!(\mathsf{double});\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 3 \oplus \{\mathtt{OK} : \epsilon, \mathtt{NOK} : \epsilon\},\\
\qquad\qquad\qquad\qquad 1!(\mathsf{money});\\
\qquad\qquad\qquad\qquad 2?(\mathsf{date})\}\\
\qquad\epsilon\}
\end{array}
$$

4.2. *Well formedness of global types with exceptions*

This subsection explains the well formedness of the global types, focusing on the conditions related to the exceptions.

4.2.1. *Linearity.* The linearity condition of global types introduced in Honda *et al.* (2008) avoids mismatches in the communication. For example,

$$\mathtt{p}_1 \to \mathtt{p}_2 : k\langle \tilde{S} \rangle ; \mathtt{p}_3 \to \mathtt{p}_4 : k\langle \tilde{S}' \rangle$$

is not a well-formed specification, because the participant $\mathtt{p}_4$ may receive the value meant for $\mathtt{p}_2$. However, if the channels are different as:

$$\mathtt{p}_1 \to \mathtt{p}_2 : k\langle \tilde{S} \rangle ; \mathtt{p}_3 \to \mathtt{p}_4 : k'\langle \tilde{S}' \rangle$$

the interaction is safe as there is no mix-up.

Linearity also ensures race-freedom in exception handling. Let us consider the following example:

$$\{k, \gamma_{\mathtt{p}_1,\mathtt{p}_2}, \gamma'_{\mathtt{p}_1,\mathtt{p}_2}\} ; \{(k,k'), \gamma_{\mathtt{p}_1,\mathtt{p}_3}, \gamma'_{\mathtt{p}_1,\mathtt{p}_3}\}$$

where $\gamma_{\mathtt{p}_i,\mathtt{p}_j}$ is the global type describing the interaction between participants $\mathtt{p}_i$ and $\mathtt{p}_j$. In this case, the second try-catch block uses the same channel s_k of the first try block. Thanks to linearity, we know that participant $\mathtt{p}_3$ will not send/receive values on that channel before the first block of interactions has terminated.

4.2.2. *Different try-catch blocks must have different sets of channels. Sequential composition.* The type system forbids two subsequent try-catch blocks on the same set of channels. For instance let us consider the following global type

$$\{k, \gamma_{\mathtt{p}_1,\mathtt{p}_2}, \gamma'_{\mathtt{p}_1,\mathtt{p}_2}\} ; \{k, \gamma_{\mathtt{p}_1,\mathtt{p}_3}, \gamma'_{\mathtt{p}_1,\mathtt{p}_3}\}$$

and the following processes where P_i is the code associated to participant $\mathtt{p}_i$:

$$
\begin{array}{l}
P_1 = \mathsf{try}(s_k)\{P_{12}\}\ \mathsf{catch}\ \{Q_{12}\}; \mathsf{try}(s_k)\{P_{13}\}\ \mathsf{catch}\ \{Q_{13}\}\\
P_2 = \mathsf{try}(s_k)\{P_{21}\}\ \mathsf{catch}\ \{Q_{21}\}\\
P_3 = \mathsf{try}(s_k)\{P_{31}\}\ \mathsf{catch}\ \{Q_{31}\}.
\end{array}
$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

If we reduce $P_1 \mid P_2 \mid P_3$ after some step (that is after having reduced the interaction between $\mathtt{p}_1$ and $\mathtt{p}_2$) we could have:

$$\begin{aligned}\Sigma \vdash\ & \mathsf{try}(s_k)\{\mathbf{0}\}\ \mathsf{catch}\ \{Q_{12}\}; \mathsf{try}(s_k)\{P_{13}\}\ \mathsf{catch}\ \{Q_{13}\} \\ \mid\ & \mathsf{try}(s_k)\{\mathbf{0}\}\ \mathsf{catch}\ \{Q_{21}\} \\ \mid\ & \mathsf{try}(s_k)\{P_{31}\}\ \mathsf{catch}\ \{Q_{31}\}.\end{aligned}$$

Now, to apply (ZThr) all try blocks on s_k must terminate before proceeding to the next try block in the sequential composition, but in this case participant $\mathtt{p}_3$ cannot perform any interaction: this is a deadlocked computation, differently from what specified in the global protocol.

This restriction on channels also holds for try-catch blocks far away in the same sequential composition, for example:

$$\{\!|(k,k'),\ \gamma_{\mathtt{p}_1,\mathtt{p}_2},\ \gamma'_{\mathtt{p}_1,\mathtt{p}_2}|\!\}; \{\!|(k',k''),\ \gamma_{\mathtt{p}_1,\mathtt{p}_3},\ \gamma'_{\mathtt{p}_1,\mathtt{p}_3}|\!\}; \{\!|(k,k'),\ \gamma_{\mathtt{p}_2,\mathtt{p}_4},\ \gamma'_{\mathtt{p}_2,\mathtt{p}_4}|\!\}$$

in this case the code of $\mathtt{p}_4$ begins with a try-catch block on channels $s_k, s_{k'}$ exactly as $\mathtt{p}_1$ and $\mathtt{p}_2$, leading to the same error as the example above.

This condition does not represent a limitation since we can always allocate distinct channels without losing expressiveness.

Parallel composition. By linearity, we cannot have more than one communication on the same channel at the same time. Extending this concept to exceptions, we require parallel try-catch blocks to have disjoint sets of channels:

$$\{\!|\tilde{k},\ \gamma_1,\ \gamma'_1|\!\} \parallel \{\!|\tilde{k}',\ \gamma_2,\ \gamma'_2|\!\} \text{ and } \tilde{k} \cap \tilde{k}' = \varnothing.$$

4.2.3. *Consistent queue levels.* As hinted in Section 3, in a try-catch block $\mathsf{try}(\tilde{s})\{P\}\ \mathsf{catch}\ \{Q\}$ if P uses channels $s_1^{\varphi_1}, \ldots, s_n^{\varphi_n}$ then Q uses $s_1^{\varphi_1+1}, \ldots, s_n^{\varphi_n+1}$. Thus in the global type $\{\!|k, \gamma, \gamma'|\!\}$, interactions within γ pass on channel s_k at level φ (and queue $s[\varphi]$) and in γ' values are exchanged using $s^{\varphi+1}$ and the queue at the level $\varphi + 1$. This is needed to deal with asynchrony of exception notification and propagation that may reach different participants in different moments: one participant may be already executing the handling code in γ', having captured an exception, while some other participant may be still behaving as prescribed by γ. Therefore, some values may be still written in the queue $s_k[\varphi]$ while some communication is already going on the next level $s_k[\varphi + 1]$ for some participant. These levels guarantee a safe interaction without out-of-date values being exchanged at the wrong moment. For example, in the following conversation:

$$\{\!|k,\ \gamma_{\mathtt{p}_1,\mathtt{p}_2},\ \gamma'_{\mathtt{p}_1,\mathtt{p}_2}|\!\}; \{\!|(k,k'),\ \gamma_{\mathtt{p}_1,\mathtt{p}_3},\ \gamma'_{\mathtt{p}_1,\mathtt{p}_3}|\!\}$$

participants $\mathtt{p}_1$ and $\mathtt{p}_2$ will communicate on channel s_k^0 within $\gamma_{\mathtt{p}_1,\mathtt{p}_2}$, and on channel s_k^1 within $\gamma'_{\mathtt{p}_1,\mathtt{p}_2}$. After a sequential composition the communication starts again on level 0, this is safe due to the garbage collection performed on the old level of the queue when an exception is raised (see rule (RThr)). Therefore, participants $\mathtt{p}_1$ and $\mathtt{p}_3$ will communicate on channel s_k^0 within $\gamma_{\mathtt{p}_1,\mathtt{p}_3}$, and on channel s_k^1 within $\gamma'_{\mathtt{p}_1,\mathtt{p}_3}$.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Channels are decorated with appropriate levels in global types since the global specification is a way for processes to acquire the knowledge concerning the communication behaviour to follow: global types also lead programmers into the procedure of updating queues in an efficient and consistent way. However, writing down queue levels into the global type may be an annoying procedure, therefore we define a function *Qlevel* that given a global type G produces an annotated global type G'. This procedure has to be performed only once before type-checking. Also in Section 2, we assumed processes to have the channels pointing at the right level of the queues at each time, without any dynamic update by the operational semantics. To this aim, we can imagine a similar function that given an annotated global type and a program, gives as a result an annotated program. Since the code must be parsed anyway for type checking, before running it, we can assume this procedure to be performed at once by the type checker.

Definition 4.1 (Qlevel).

$$Qlevel(\gamma;\mathsf{end}) = qlev(\gamma,\tilde{k} : \tilde{0}),\ \text{where } \tilde{k} = \mathsf{ch}(\gamma)$$
$$qlev(\mathtt{p}_1 \to \mathtt{p}_2 : k\langle\tilde{S}\rangle,\tilde{k} : \tilde{\varphi},k : \varphi) = \mathtt{p}_1 \to \mathtt{p}_2 : k^{\varphi}\langle\tilde{S}\rangle$$
$$qlev(\mathtt{p}_1 \to \mathtt{p}_2 : k\{l_i : \gamma_i\}_{i\in I},\tilde{k} : \tilde{\varphi},k : \varphi) = \mathtt{p}_1 \to \mathtt{p}_2 : k^{\varphi}\{l_i : qlev(\gamma_i,\tilde{k} : \tilde{\varphi},k : \varphi)\}_{i\in I}$$
$$qlev(\{\!|\tilde{k},\gamma,\gamma'|\!\},\tilde{k}' : \tilde{\varphi}',\tilde{k} : \tilde{\varphi}) = \{\!|\tilde{k}^{\tilde{\varphi}}, qlev(\gamma,\tilde{k} : \tilde{\varphi}), qlev(\gamma',\tilde{k} : (\tilde{\varphi}+\tilde{\delta}+1))|\!\}$$
$$\text{where } \delta_i = maxlev(\gamma,k_i : \varphi_i)$$
$$qlev(\gamma;\gamma',\tilde{k} : \tilde{\varphi}) = qlev(\gamma,\tilde{k} : \tilde{\varphi});qlev(\gamma',\tilde{k} : \tilde{\varphi})$$
$$qlev(\gamma \parallel \gamma',\tilde{k} : \tilde{\varphi}) = qlev(\gamma,\tilde{k} : \tilde{\varphi}) \parallel qlev(\gamma',\tilde{k} : \tilde{\varphi})$$
$$qlev(\mu\mathbf{t}.\gamma,\tilde{k} : \tilde{\varphi}) = \mu\mathbf{t}.qlev(\gamma,\tilde{k} : \tilde{\varphi}) \qquad qlev(\mathbf{t},\tilde{k} : \tilde{\varphi}) = \mathbf{t} \qquad qlev(\epsilon,\tilde{k} : \tilde{\varphi}) = \epsilon$$

where $maxlev(\gamma,k : \varphi)$ is the maximum index given to k by $qlev(\gamma,k : \varphi)$.

Qlevel is applied to global types by using the auxiliary function *qlev* on partial types which at the beginning put all the channels in the global type at level 0. The level of channels is increased when *qlev* is applied to a try-catch type: $qlev(\{\!|\tilde{k},\gamma,\gamma'|\!\},\tilde{k}' : \tilde{\varphi}',\tilde{k} : \tilde{\varphi})$. First let us notice that the set of channels involved in the current try block ($\tilde{k}$) is a subset of the channels in the sessions ($\tilde{k} \cup \tilde{k}'$). *qlev* assigns level φ to $\tilde{k}$, then it is applied to the type of the default process and of the handler ($qlev(\gamma,\tilde{k} : \tilde{\varphi})$ and $qlev(\gamma',\tilde{k} : (\tilde{\varphi}+\tilde{\delta}+1))$ respectively) by taking into account the number of nested try blocks in which the channels are involved. As an example, consider:

$$\{\!|(k,k',k''),(\{\!|k,\gamma_1,\gamma_1'|\!\};\{\!|(k,k'),\gamma_2,\gamma_2'|\!\}),\gamma|\!\}.$$

Here, we have that in γ_1 communications are carried on s_k^0, within γ_1' we have s_k^1. After the sequential composition the numbering starts again from 0, thus in γ_2 communications are carried on s_k^0 and $s_{k'}^0$, within γ_2' we have s_k^1 and $s_{k'}^1$. Finally within γ the communication are on channels s_k^2, $s_{k'}^2$, and $s_{k''}^0$, since $maxlev(\{\!|k,\gamma_1,\gamma_1'|\!\};\{\!|k,k',\gamma_2,\gamma_2'|\!\},k : 0) = 1$, $maxlev(\{\!|k,\gamma_1,\gamma_1'|\!\};\{\!|k,k',\gamma_2,\gamma_2'|\!\},k' : 0) = 1$, and $maxlev(\{\!|k,\gamma_1,\gamma_1'|\!\};\{\!|k,k',\gamma_2,\gamma_2'|\!\},k'' : 0) = 0$. Notice that, since no try-catch construct can occur underneath recursion (see Section 4.2.4) the level of channels in a recursive process will remain unchanged at each recursion. This is reflected by the application of *qlev* on a recursive type $\mu\mathbf{t}.\gamma$, which produces a recursive type that is still not distinguishable from the result produced if applied on the equivalent type $\gamma\{\mu\mathbf{t}.\gamma/\mathbf{t}\}$.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

4.2.4. *No try-catch construct underneath recursion.* Let us consider the following global type

$$\mu \mathbf{t}.\{\tilde{k},\ \gamma,\ \gamma';\mathbf{t}\} \tag{1}$$

where γ and $\gamma';\mathbf{t}$ describe default and exceptional interaction between the two participants p_1 and p_2. Suppose the process associated to p_1 is:

$$P_1 \ = \ \mathsf{def}\ X(s_k) = \overbrace{\mathsf{try}(s_k)\{P \mid \mathsf{throw}(s_k)\}\ \mathsf{catch}\ \{s_k^1!\langle 1\rangle; X\left\langle s_k^1\right\rangle\}}^{R}\ \mathsf{in}\ X\langle s_k\rangle$$

and P_2 the one associated to p_2. Then if we reduce:

$$\varnothing \vdash \mathsf{def}\ X(s_k) = \mathsf{try}(s_k)\{P \mid \mathsf{throw}(s_k)\}\ \mathsf{catch}\ \{s_k^1!\langle 1\rangle; X\left\langle s_k^1\right\rangle\}\ \mathsf{in} X\langle s_k\rangle \ \mid P_2 \mid s_k[0] : \varnothing$$

by applying rules ⌊Def⌋, ⌊Thr⌋, ⌊RThr⌋ and ⌊Send2⌋ we obtain:

$$\begin{aligned}\mathsf{throw}(s_k) \vdash\ & \mathsf{def}\ X(s_k) = R\ \mathsf{in}\ \mathsf{try}(s_k^1)\{P' \mid \mathsf{throw}(s_k^1)\}\ \mathsf{catch}\ \{s_k^1!\langle 1\rangle; X\left\langle s_k^1\right\rangle\} \mid \\ & P_2' \mid s_k[0] : \varnothing \mid s_k[1] : 1\end{aligned}$$

where both P' and P_2' use channel s_k^1 to interact: we can have communication mismatches due to the fact that both handlers and default processes use the same channel at the same level.

Similar mismatches may be encountered when the recursion variable occurs inside the default process, as described by the following global specification:

$$\mu \mathbf{t}.\{\tilde{k},\ \gamma;\mathbf{t},\ \gamma'\} \tag{2}$$

which enables infinite chains of try-catch constructs. Notice that (2) would also break the *atomicity* requirement (see next Section 4.2.5), since nested exceptions must affect a strictly smaller set of channels, while (2) is equivalent to a try-catch construct with multiple nested try-catch on the same set of channels.

Further problems arise when the recursion variable is outside the try-catch construct but still such a construct occurs somewhere underneath the recursion. For instance in

$$\mu \mathbf{t}.\{\tilde{k},\ \gamma,\ \gamma'\};\mathbf{t} \tag{3}$$

we will end up having multiple occurrences of the same channels at the same level in different places in the global exception hierarchy. Such a configuration may lead to an analogous communication misbehaviour as the one depicted above. Notice that (3) would also break the *sequential composition* requirements, and similarly $\mu \mathbf{t}.\{\tilde{k},\ \gamma,\ \gamma'\} \parallel \mathbf{t}$ would break the *parallel composition* requirements (see Section 4.2.2). To avoid these problems we forbid try-catch constructs to occur underneath recursion. This decision also reflects the nature of global exceptions: the default communication is carried out following the try-block's prescription, once an exception is raised the execution is moved to *another* step of the interaction, described in the catch-block, which represents a new conversation reached by all participants in a concerted way. Having a try block underneath the recursion would mean restarting over the same communication pattern with the same channel instantiation. We consider moving the recursion within the try-block a better way to implement that recurring communication behaviour, indeed most of the interesting examples and use cases can be safely described as $\{\tilde{k},\ \mu \mathbf{t}.\gamma,\ \mu \mathbf{t}'.\gamma'\}$.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$(\gamma;\mathsf{end}) \upharpoonright \mathtt{p} = (\gamma \upharpoonright \mathtt{p});\mathsf{end}$$

$$(\mathtt{p}_1 \to \mathtt{p}_2 : k^{\varphi}\langle \tilde{S}\rangle) \upharpoonright \mathtt{p} = \begin{cases} k^{\varphi}!(\tilde{S}) & \text{if } \mathtt{p} = \mathtt{p}_1, \\ k^{\varphi}?(\tilde{S}) & \text{if } \mathtt{p} = \mathtt{p}_2, \\ \epsilon & \text{otherwise.} \end{cases}$$

$$(\mathtt{p}_1 \to \mathtt{p}_2 : k^{\varphi}\{l_i : \gamma_i\}_{i\in I}) \upharpoonright \mathtt{p} = \begin{cases} k^{\varphi} \oplus \{l_i : \gamma_i \upharpoonright \mathtt{p}\}_{i\in I} & \text{if } \mathtt{p} = \mathtt{p}_1 \\ k^{\varphi} \& \{l_i : \gamma_i \upharpoonright \mathtt{p}\}_{i\in I} & \text{if } \mathtt{p} = \mathtt{p}_2 \\ \gamma_1 \upharpoonright \mathtt{p} & \text{if } \mathtt{p} \neq \mathtt{p}_1, \mathtt{p} \neq \mathtt{p}_2 \\ & \text{and } \gamma_i \upharpoonright \mathtt{p} = \gamma_j \upharpoonright \mathtt{p} \text{ for all } i, j \in I. \end{cases}$$

$$(\{\tilde{k}, \gamma_1, \gamma_2\}) \upharpoonright \mathtt{p} = \begin{cases} \{\tilde{k}, (\gamma_1 \upharpoonright \mathtt{p}), (\gamma_2 \upharpoonright \mathtt{p})\} & \text{if } \mathsf{ch}(\gamma_i) \subseteq \tilde{k}, \text{ and } (\gamma_1 \upharpoonright \mathtt{p} \neq \epsilon \text{ or } \gamma_2 \upharpoonright \mathtt{p} \neq \epsilon) \\ & \text{and } \left(\{\tilde{k}', \gamma_1', \gamma_2'\} \in \gamma_1 \text{ implies } \tilde{k}' \subset \tilde{k}\right), \\ \epsilon & \text{if } \mathsf{ch}(\gamma_i) \subseteq \tilde{k}, \text{ and } \gamma_i \upharpoonright \mathtt{p} = \epsilon \\ & \text{and } \left(\{\tilde{k}', \gamma_1', \gamma_2'\} \in \gamma_1 \text{ implies } \tilde{k}' \subset \tilde{k}\right), \end{cases}$$

$$(\gamma_1 \parallel \gamma_2) \upharpoonright \mathtt{p} = \begin{cases} \gamma_i \upharpoonright \mathtt{p} & \text{if } \mathtt{p} \in \gamma_i \text{ and } \mathtt{p} \notin \gamma_j, i \neq j \in \{1,2\} \\ \mathsf{end} & \text{otherwise.} \end{cases}$$

$$(\gamma_1;\gamma_2) \upharpoonright \mathtt{p} = (\gamma_1 \upharpoonright \mathtt{p});(\gamma_2 \upharpoonright \mathtt{p}) \qquad \mathbf{t} \upharpoonright \mathtt{p} = \mathbf{t} \qquad (\mu \mathbf{t}.\gamma) \upharpoonright \mathtt{p} = \begin{cases} \mu \mathbf{t}.(\gamma \upharpoonright \mathtt{p}) & \text{if } \mathtt{p} \text{ occurs in } \gamma \\ \mathsf{end} & \text{otherwise} \end{cases}$$

Fig. 8. Projection.

4.2.5. *Atomicity*. Our model of distributed global escapes is based on the idea, (supported by practical industrial examples as the one in Section 6), that local exceptions, if can be handled, must be confined within involved participants, without notifying enclosing components. Operational semantics assumes that enclosed exceptions must be on smaller sets of channels. This assumption is enforced by the global type, where the same condition holds: inner exceptions involve less channels than the outer ones. The projection function (see Section 4.3) discharges global types that do not satisfy this constraint, automatically ensuring atomicity.

4.3. *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 $\mathtt{p}$ (Figure 8). We write $G \upharpoonright \mathtt{p}$ to denote such a projection. Notice that we apply projection to annotated global types.

The projection is defined first over global types and then over partial global types, see Figure 8.

The only new projection rule is the one for the exception type. This type is projected in every participant as a local exception type, only if the participant has some activity in the try-catch block and if the try-catch blocks enclosed in the default protocol γ_1 are defined on a smaller set of channels. Notice that we do not check anything for the try blocks possibly occurring in the handler: this is because they must be on a smaller set of channels w.r.t. a possible outer try block, then this condition has been checked at the outer step. For instance, for the global type $\{\tilde{k}, \{\tilde{k}', \{\tilde{k}'', P'', Q''\}, \{\tilde{k}''', P''', Q'''\}\}, Q\};\mathsf{end}$ to be projectable the following conditions must hold:

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

i. $\tilde{k}' \subset \tilde{k}$, because $\{\tilde{k}', \{\tilde{k}'', P'', Q''\}, \{\tilde{k}''', P''', Q'''\}\}$ is a try catch occurring in the default global protocol of the outer try-catch block on channels $\tilde{k}$;
ii. $\tilde{k}'' \subset \tilde{k}' (\subset \tilde{k})$, because $\{\tilde{k}'', P'', Q''\}$ is a try catch occurring in the default global protocol of the try-catch block on channels $\tilde{k}'$ (and also in the default protocol of the outer try catch on channels $\tilde{k}$);
iii. $k''' \subset k$, because also $\{\tilde{k}''', P''', Q'''\}$ is a try catch occurring in the default global protocol of the outer try-catch block on channels $\tilde{k}$;

Notice that no condition is required for $\tilde{k}'''$ w.r.t. $\tilde{k}'$.
When the side condition does not hold the mapping is undefined.

Remark 1. One may notice that with our definitions, sequentiality may not be respected: for instance the projection of $\gamma_1 ; \gamma_2$ with respect to a participant $\mathtt{p}$ may be the same as the projection of $\gamma_1 \parallel \gamma_2$ if $\mathtt{p}$ has some activity only in γ_2. This is because $\mathtt{p}$ has no knowledge of the status of the interactions within γ_1 and can perform a sending action before all the interactions in γ_1 are completed. In our model this is not an error, because it cannot lead to a mismatch, we simply consider that the sequence has the same behaviour as the parallel composition, for those participants that are involved only in γ_1 or γ_2. We could have been more restrictive and forbid this kind of specifications but we do not need it for our goal, communication safety. On the other hand, thanks to semantic rule (ZThr) sequentiality among exception blocks is always respected.

4.4. *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 $\Gamma ::= \varnothing \mid \Gamma, u : S \mid \Gamma, X : \tilde{S}\tilde{A}$
— the *session environment* Δ that records session types associated to session channels and is defined by $\Delta ::= \varnothing \mid \Delta, \tilde{k} : \{A@\mathtt{p}\}_{\mathtt{p} \in I}$.

The typing judgement has the form $\Gamma \vdash P \triangleright \Delta$ for processes and $\Gamma \vdash e$ for expressions and the complete set of typing rules is given in Figure 9.

All the rules are standard and as the same as original rules in Honda *et al.* (2008) except for the two rules (Try) and (Throw).

(Name), (Bool) and (Or) are standard rules for expressions. (Inact) and (NRes) are standard rules for inaction and restriction.

(MReq) is the rule for the session request. The type for $\tilde{s}$ is the first projection of the declared global type for a in Γ. (Macc) is for session accept by the $\mathtt{p}$-th participant. The end-point type $(G \upharpoonright \mathtt{p})@\mathtt{p}$ means that the participant $\mathtt{p}$ has end-point type $G \upharpoonright \mathtt{p}$, which is the projection of G onto $\mathtt{p}$. The condition $|\tilde{s}| = |\mathsf{sid}(G)|$ ensures the number of session channels meets those in G.

(Send) and (Rcv) are standard rules for input/output of values. Since the kth name with level φ s_k^{φ} of $\tilde{s}$ is used as the subject, we record the number k^{φ}.

(Sel) and (Branch) are the rules for selection and branching.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$\text{(Name)}\ \Gamma, a : S \vdash a : S \qquad \text{(Bool)}\ \Gamma \vdash \mathsf{true}, \mathsf{false} : \mathit{bool} \qquad \text{(Or)}\ \frac{\Gamma \vdash e_i : \mathit{bool}}{\Gamma \vdash e_1 \,\mathtt{or}\, e_2 : \mathit{bool}} \qquad \text{(Inact)}\ \frac{\Delta\ \mathsf{end\ only}}{\Gamma \vdash \mathbf{0} \triangleright \Delta} \qquad \text{(NRes)}\ \frac{\Gamma, a : \langle G \rangle \vdash P \triangleright \Delta}{\Gamma \vdash (\nu a)P \triangleright \Delta}$$

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

$$\text{(MAcc)}\ \frac{\Gamma \vdash a : \langle G \rangle \quad \Gamma \vdash P \triangleright \Delta \cup \{\tilde{s} : (G \upharpoonright \mathtt{p})@\mathtt{p}\} \quad |\tilde{s}| = |\mathsf{sid}(G)|}{\Gamma \vdash a[\mathtt{p}](\tilde{s}).P \triangleright \Delta}$$

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

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

$$\text{(Try)}\ \frac{\Gamma \vdash P \triangleright \{\tilde{s} : \alpha@\mathtt{p}\} \quad \Gamma \vdash Q \triangleright \{\tilde{s} : \beta@\mathtt{p}\} \quad \mathsf{ch}(P) \subseteq \tilde{z}}{\Gamma \vdash \mathsf{try}(\tilde{z})\{P\}\ \mathsf{catch}\ \{Q\} \triangleright \{\tilde{s} : \{\tilde{z}, \alpha, \beta\}@\mathtt{p}\}} \qquad \text{(Throw)}\ \Gamma \vdash \mathsf{throw}(\tilde{r}) \triangleright \Delta$$

$$\text{(Par)}\ \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta' \quad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q \triangleright \Delta \circ \Delta'} \qquad \text{(Seq)}\ \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta'}{\Gamma \vdash P; Q \triangleright \Delta \cdot \Delta'} \qquad \text{(If)}\ \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash Q \triangleright \Delta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleright \Delta}$$

$$\text{(Var)}\ \frac{\Gamma \vdash \tilde{e} : \tilde{S} \qquad \Delta\ \mathsf{end\ only}}{\Gamma, X : \tilde{S}\tilde{A} \vdash X\langle \tilde{e}\tilde{s}_1..\tilde{s}_n \rangle \triangleright \Delta, \tilde{s}_1 : A_1@\mathtt{p}_1, .., \tilde{s}_n : A_n@\mathtt{p}_n}$$

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

Fig. 9. Typing rules.

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.

In addition, the rule 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 (Throw) the process $\mathsf{throw}(\tilde{r})$ is typed with any session environment.

The rule for parallel composition uses the concepts of parallel composition of session types and environment, and of compatibility between session environments. The definitions, which are standard, follow.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Definition 4.2 (parallel composition of session types). Session types can be composed in the following way: $\{A_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I} \circ \{A'_{\mathtt{p}'}@\mathtt{p}'\}_{\mathtt{p}'\in J} = \{A_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p}\in I} \cup \{A'_{\mathtt{p}'}@\mathtt{p}'\}_{\mathtt{p}'\in J}$, if $I \cap J = \emptyset$.

Definition 4.3 (compatibility between session environments). $\Delta_1 \asymp \Delta_2$, denotes the *compatibility* between Δ_1 and Δ_2, if for all $\tilde{s}_i \in dom(\Delta_i)$ such that $\tilde{s}_1 \cap \tilde{s}_2 \neq \emptyset$, $\tilde{s} = \tilde{s}_1 = \tilde{s}_2$ and $\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s})$ is defined.

Definition 4.4 (parallel composition of session environments). The *parallel composition of two compatible environments* Δ_1 *and* Δ_2, written $\Delta_1 \circ \Delta_2$, is given as:

$$\Delta_1 \circ \Delta_2 \;=\; \{\Delta_1(\tilde{s}) \circ \Delta_2(\tilde{s}) \mid \tilde{s} \in dom(\Delta_1) \cap dom(\Delta_2)\} \;\cup\; \Delta_1 \setminus dom(\Delta_2) \cup \Delta_2 \setminus dom(\Delta_1)$$

The following definition of sequential composition of session environments is used in rule (Seq).

Definition 4.5 (sequential composition of session environments). $\Delta_1 \cdot \Delta_2$ denotes the sequential composition of the environments and it is defined as follows:

$$\Delta_1 \cdot \Delta_2(\tilde{s}) = \begin{array}{ll} A@\mathtt{p} & \text{if } \Delta_1(\tilde{s}) = A@\mathtt{p} \text{ and } \Delta_2(\tilde{s}) = \epsilon@\mathtt{p} \\ \alpha; A@\mathtt{p} & \text{if } \Delta_1(\tilde{s}) = \alpha@\mathtt{p} \text{ and } \Delta_2(\tilde{s}) = A@\mathtt{p} \\ \Delta_1(\tilde{s}) & \text{if } \Delta_1(\tilde{s}) \text{ defined and } \Delta_2(\tilde{s}) \text{ undefined} \\ \Delta_2(\tilde{s}) & \text{if } \Delta_1(\tilde{s}) \text{ undefined and } \Delta_2(\tilde{s}) \text{ defined} \\ \bot & \text{otherwise.} \end{array}$$

(If) , (Var) and (Def) are standard rules for conditional, process call and recursion. Note that given closed annotated processes (i.e. bound variables and names are annotated by types), the type checking is decidable (this can be proved by using the complexity results in Deniélou and Yoshida (2010)).

5. Properties

The type discipline ensures:

— the lack of standard type errors in expressions (*subject reduction theorem*, see Theorem 5.8);
— communication error freedom (*communication safety theorem*, see Theorem 5.9);
— that the interactions of a typable process exactly follow the specification described by its global type (*session fidelity*, see Corollary 5.10);
— that, if interactions within a session are not hindered by initialization and communication of different sessions, then the converse holds: the reduction predicted by the typing surely takes place (*progress theorem*, see Theorem 5.15);
— *Termination property*, see Corollary 5.16, that states that a well-formed process $P\,;Q$ eventually reduces to Q, provided that P does not contain any infinite recursion;
— *Throw confluence*, see Theorem 5.17, that guarantees the atomicity of our exceptions, stating that if an exception on some channels $\tilde{r}$ is thrown, then all communications on some other $r' \notin \tilde{r}$ can go on concurrently without interferences and if some action on $r \in \tilde{r}$ can be performed then there is no mismatch on the exchanged data if the action is performed before the exception has been captured.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$\frac{\Gamma \vdash P \triangleright_{\tilde{r}} \Delta \quad \Delta \leq \Delta'}{\Gamma \vdash P \triangleright_{\tilde{r}} \Delta'} \qquad \frac{\Delta \text{ end only}}{\Gamma \vdash s_k[\varphi] : \emptyset \ \triangleright_{s_k^{\varphi}} \tilde{s} : \{[\ ]@\mathtt{p}\}_{\mathtt{p}} \circ \Delta} \qquad (\textsc{Subs}),(\textsc{Qnil})$$

$$\frac{\Gamma \vdash v_i : S_i \quad \Gamma \vdash s_k[\varphi] : L \triangleright_{s_k^{\varphi}} \Delta, \tilde{s} : (\{\mathcal{T}@\mathtt{q}\} \cup R) \quad R = \{H_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I}}{\Gamma \vdash s_k[\varphi] : (L :: \tilde{v}) \ \triangleright_{s_k^{\varphi}} \ \Delta, \tilde{s} : \{\mathcal{T}[k^{\varphi}!\langle \tilde{S} \rangle; [\ ]]@\mathtt{q}\} \cup R)} \qquad (\textsc{Qval})$$

$$\frac{\Gamma \vdash s_k[\varphi] : L \triangleright_{s_k^{\varphi}} \Delta, \tilde{s} : \{\mathcal{T}@\mathtt{q}\} \cup R \quad R = \{H_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I}}{\Gamma \vdash s_k[\varphi] : (L :: l) \ \triangleright_{s_k^{\varphi}} \Delta, \tilde{s} : \{\mathcal{T}[k^{\varphi} \oplus l : [\ ]]@\mathtt{q}\} \cup R} \qquad (\textsc{Qsel})$$

$$\frac{\Gamma \vdash P \triangleright_{\tilde{r}} \Delta \quad \Gamma \vdash Q \triangleright_{\tilde{r}'} \Delta' \quad \tilde{r} \cap \tilde{r}' = \emptyset \quad \Delta \asymp \Delta'}{\Gamma \vdash P \mid Q \triangleright_{\tilde{r}, \tilde{r}'} \Delta \circ \Delta'} \qquad (\textsc{Conc})$$

$$\frac{\Gamma \vdash P \triangleright_{\tilde{r}} \Delta, \tilde{r}' : \{T_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I} \quad \tilde{r}' \in \tilde{r} \quad \{T_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I} \text{ coherent}}{\Gamma \vdash (\boldsymbol{\nu}\, \tilde{r}')P \triangleright_{\tilde{r} \setminus \tilde{r}'} \ \Delta} \qquad (\textsc{CRes})$$

Fig. 10. Selected typing rules for runtime processes.

In this section, we just show main results, while proofs and auxiliary definitions can be found in Appendix C. We want a global type to have well-defined projections, then we use the concept of coherence. We define $\mathsf{pid}(G)$ the set of participant numbers occurring in G.

Definition 5.1 (coherence). (1) We say G is *coherent* if it is linear and $G \upharpoonright \mathtt{p}$ is well defined for each $\mathtt{p} \in \mathsf{pid}(G)$, similarly for each carried global type inductively. (2) $\{A_{\mathtt{p}}@\mathtt{p}\}_{\mathtt{p} \in I}$ is *coherent* if for some coherent G such that $I = \mathsf{pid}(G)$, we have $G \upharpoonright \mathtt{p} = A_{\mathtt{p}}$ for each $\mathtt{p} \in I$.

Definition 5.2 (type contexts). The type contexts $(\mathcal{T}, \mathcal{T}', \ldots)$ and the extended session typing $(\Delta, \Delta', \ldots$ as before) are given as:

$$\mathcal{T} \ ::= \ [\ ] \mid k^{\varphi}!\langle S \rangle; \mathcal{T} \mid k^{\varphi} \oplus l_i : \mathcal{T} \qquad H \ ::= \ A \mid \mathcal{T} \qquad \Delta \ ::= \ \varnothing \mid \Delta, \tilde{s} : \{H_{\mathtt{p}}\}_{\mathtt{p} \in I}$$

5.1. *Typing rules for runtime*

To guarantee that there is at most one queue for each channel, we use the typing judgement refined as:

$$\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$$

where $\tilde{r} = s_1^{\varphi_1}, \ldots, s_n^{\varphi_n}$ (regarded as a set) records the session channels (with their level φ_i) associated with the message queues. The typing rules for runtime are given in Figure 10.

(Subs) allows subsumption ($\leqslant_{\mathsf{sub}}$ is extended pointwise from types. See Definition C.2). (Qnil) starts from the empty hole for each participant, recording the session channel in the judgement. (Qval) says that when we enqueue $\tilde{v}$, the type for $\tilde{v}$ is added to the tail. (Qsel) is the corresponding rules for labels. (Conc) is refined to prohibit duplicated message queues. The rule (Conc) uses the partial operator $\circ$, defined as follows:

$$A \circ \mathcal{T} = \mathcal{T} \circ A = \mathcal{T}[A] \qquad \mathcal{T} \circ \mathcal{T}' = \mathcal{T}[\mathcal{T}'] \ (\text{if } \mathsf{sid}(\mathcal{T}) \cap \mathsf{sid}(\mathcal{T}') = \varnothing),$$

where $\mathsf{sid}(\mathcal{T})$ denotes the channel numbers in $\mathcal{T}$.

We extend $\Delta \asymp \Delta'$ and $\Delta \circ \Delta'$ for the above extensions. In (CRes), since we are hiding session channels, we now know no other participants can be added. Hence, we check all message queues are composed and the given configuration at $\tilde{s}$ is coherent.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$k^{\varphi}!\langle S\rangle; H@\mathtt{p}, k^{\varphi}?\langle S\rangle; T@\mathtt{q} \xrightarrow{k^{\varphi}} H@\mathtt{p}, T@\mathtt{q} \qquad \text{(TR-Com)}$$

$$\{\tilde{r},\ k^{\varphi}!\langle S\rangle; H,\ H'\}@\mathtt{p}, \{\tilde{r},\ k^{\varphi}?\langle S\rangle; T,\ H''\}@\mathtt{q} \xrightarrow{k^{\varphi}} \{\tilde{r},\ H,\ H'\}@\mathtt{p}, \{\tilde{r},\ T,\ H''\}@\mathtt{q} \qquad \text{(TR-ComEx)}$$

$$k^{\varphi} \oplus \{l : H, ...\}@\mathtt{p}, k^{\varphi}\&\{l : T, ...\}@\mathtt{q} \xrightarrow{k^{\varphi}} H@\mathtt{p}, T@\mathtt{q} \qquad \text{(TR-Bra)}$$

$$\{\tilde{k^{\varphi}},\ k^{\varphi} \oplus \{l : H, ...\},\ H'\}@\mathtt{p}, \{\tilde{k^{\varphi}},\ k^{\varphi}\&\{l : T, ...\},\ H''\}@\mathtt{q} \xrightarrow{k^{\varphi}} \{\tilde{k^{\varphi}},\ H,\ H'\}@\mathtt{p}, \{\tilde{k^{\varphi}},\ T,\ H''\}@\mathtt{q} \qquad \text{(TR-BraEx)}$$

$$\{\tilde{k^{\varphi}},\ H,\ H'\}@\mathtt{p} \xrightarrow{\mathsf{throw}(\tilde{k^{\varphi}})} H'@\mathtt{p} \qquad \text{(TR-Thr)}$$

$$\frac{H_1@\mathtt{p}_1, H_2@\mathtt{p}_2 \xrightarrow{k^{\varphi}} H_1'@\mathtt{p}_1, H_2'@\mathtt{p}_2 \quad 1,2 \in I}{\tilde{s} : \{H_1@\mathtt{p}_1, H_2@\mathtt{p}_2, ...\}_{i\in I}, \Delta \xrightarrow{s_k^{\varphi}} \tilde{s} : \{H_1'@\mathtt{p}_1, H_2'@\mathtt{p}_2, ...\}_{i\in I}, \Delta} \qquad \text{(TR-Context-1)}$$

$$\frac{H_1@\mathtt{p}_1 \xrightarrow{\mathsf{throw}(\tilde{k^{\varphi}})} H_1'@\mathtt{p}_1}{\tilde{s} : \{H_1@\mathtt{p}_1, ...\}_{i\in I}, \Delta \xrightarrow{\mathsf{throw}(\tilde{s_k^{\varphi}})} \tilde{s} : \{H_1'@\mathtt{p}_1, ...\}_{i\in I}, \Delta} \qquad \text{(TR-Context-2)}$$

$$\frac{\begin{array}{c} H_i'@\mathtt{p}_i, H_j'@\mathtt{p}_j \xrightarrow{k^{\varphi}} H_i''@\mathtt{p}_i, H_j''@\mathtt{p}_j \\ i,j \in \{1,\ldots,m+n\} \quad H_{n+1}', \ldots, H_{n+m}' \neq \{\tilde{k^{\varphi}},\ \mathbf{0},\ H\} \end{array}}{\begin{array}{l} \tilde{s} : \{\{\tilde{k^{\varphi}},\ \mathbf{0},\ H_1\}; H_1'@\mathtt{p}_1, \ldots, \{\tilde{k^{\varphi}},\ \mathbf{0},\ H_n\}; H_n'@\mathtt{p}_n, H_{n+1}'@\mathtt{p}_{n+1}, \ldots, H_{n+m}'@\mathtt{p}_{n+\mathtt{m}}\}, \Delta \\ \xrightarrow{s_k^{\varphi}} \tilde{s} : \{H_1'@\mathtt{p}_1, \ldots, H_i''@\mathtt{p}_i, \ldots, H_j''@\mathtt{p}_j, \ldots, H_{n+m}'@\mathtt{p}_{n+\mathtt{m}}\}, \Delta \end{array}} \qquad \text{(TR-ZeroEx)}$$

$$\frac{\Delta \approx \Delta_1 \quad \Delta_1 \xrightarrow{\lambda} \Delta_2 \quad \Delta_2 \approx \Delta'}{\Delta \xrightarrow{\lambda} \Delta'} \qquad \text{(TR-Iso)}$$

Fig. 11. Reduction over session typings and global types.

The original typing rules in Figure 9 not appearing in Figure 10 are refined as follows: (MAcc), (MReq), (Try), (Seq), (Send),(Rcv),(Sel),(Branch) replace $\Gamma \vdash P \triangleright \Delta$ with $\Gamma \vdash P \triangleright_{\varnothing} \Delta$, (Def), (Inact) allows weakening for empty queue types and (NRes)replaces $\Gamma \vdash P \triangleright \Delta$ by $\Gamma \vdash P \triangleright_{\tilde{s}} \Delta$.

Proposition 5.3. If $\Gamma \vdash P \triangleright_{s_1^{\varphi_1}..s_m^{\varphi_n}} \Delta$ then P has a unique queue at $s_i^{\varphi_j}$ $(1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n)$, no other queue at a free channel occurs in P, and no queue in P is under any prefix.

5.2. *Type reduction*

Reduction over session typings and global types (Figure 11) abstractly represents interaction at session channels. We assume all typing environments are well formed.

In the following, we use λ to range over type reduction labels, namely $\lambda ::= k^{\varphi} \mid \mathsf{throw}(\tilde{k^{\varphi}})$.

Assume G is coherent. Then full projection of G, denoted by $[\![G]\!]$ is defined as the family $\{(G \upharpoonright \mathtt{p})@\mathtt{p} \mid \mathtt{p} \in \mathsf{pid}(G)\}$.

In order to extend the typing reduction to global types, first we need to define prefixes and prefix ordering.

Definition 5.4 (prefix and carried types). We say the initial '$\mathtt{p} \to \mathtt{p}' : k$' in $\mathtt{p} \to \mathtt{p}' : k\langle S\rangle.G'$ and $\mathtt{p} \to \mathtt{p}' : k\{l_j : G_j\}_{j\in J}$ is a *prefix from* p *to* p′ *at* k *over* G' where in the former S is

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

called a *carried type*. If S is a carried type in a prefix in G then S is also a carried type in G.

Definition 5.5 (prefix ordering). Let $\mathsf{n}_1, \mathsf{n}_2$ be prefixes occurring in a global type G, i.e., $\mathsf{n}_1, \mathsf{n}_2 \in G$ (but not in its carried types). Then we write $\mathsf{n}_1 \prec \mathsf{n}_2 \in G$ when n_1 directly or indirectly prefixes n_2 in G. Formally $\prec$ is the least partial order including:

$$\begin{array}{ll} \mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k\,\langle S\rangle.G' & \text{if } \mathsf{n}_1 = \mathtt{p} \to \mathtt{p}' : k, \mathsf{n}_2 \in G' \\ \mathsf{n}_1 \prec \mathsf{n}_2 \in \mathtt{p} \to \mathtt{p}' : k\,\{l_j : G_j\}_{j\in J} & \text{if } \mathsf{n}_1 = \mathtt{p} \to \mathtt{p}' : k, \exists i \in J.\, \mathsf{n}_2 \in G_i \\ \mathsf{n}_1 \prec \mathsf{n}_2 \in \gamma; G' & \text{if } \mathsf{n}_1 \in \gamma,\ \mathsf{n}_2 \in G' \end{array}$$

Notice that, according to the previous definition, given $G = \gamma; \{\!|\tilde{k}^{\tilde{\varphi}}, \gamma_1, \gamma_2|\!\}; G'$, if $\mathsf{n}_1 \in \gamma_1$ and $\mathsf{n}_2 \in \gamma_2$, then there is no ordering relation between the two, while it holds that $\mathsf{n} \prec \mathsf{n}_1$ and $\mathsf{n} \prec \mathsf{n}_2$ for some $\mathsf{n} \in \gamma$, and it holds as well that $\mathsf{n}_1 \prec \mathsf{n}'$ and $\mathsf{n}_2 \prec \mathsf{n}'$, for some $\mathsf{n}' \in G'$.

Definition 5.6 (dependency relations). Fix G. The relation $\prec_\chi$, with $\chi \in \{\mathsf{II}, \mathsf{IO}, \mathsf{OO}\}$, over its prefixes is generated from:

$$\begin{array}{l} \mathsf{n}_1 \prec_{\mathsf{II}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2 \text{ and } \mathsf{n}_i = \mathtt{p}_i \to \mathtt{p} : k_i \ (i = 1, 2) \\ \mathsf{n}_1 \prec_{\mathsf{IO}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2, \mathsf{n}_1 = \mathtt{p}_1 \to \mathtt{p} : k_1 \text{ and } \mathsf{n}_2 = \mathtt{p} \to \mathtt{p}_2 : k_2. \\ \mathsf{n}_1 \prec_{\mathsf{OO}} \mathsf{n}_2 \text{ if } \mathsf{n}_1 \prec \mathsf{n}_2, \mathsf{n}_i = \mathtt{p} \to \mathtt{p}_i : k \ (i = 1, 2) \end{array}$$

An *input dependency from n_1 to n_2* is a chain of the form $\mathsf{n}_1 \prec_{\chi_1} \cdots \prec_{\chi_n} \mathsf{n}_2$ ($n \geqslant 0$) such that $\chi_i \in \{\mathsf{II}, \mathsf{IO}\}$ for $1 \leqslant i \leqslant n-1$ and $\chi_n = \mathsf{II}$. An *output dependency from n_1 to n_2* is a chain $\mathsf{n}_1 \prec_{\chi_1} \cdots \prec_{\chi_n} \mathsf{n}_2$ ($n \geqslant 1$) such that $\chi_i \in \{\mathsf{OO}, \mathsf{IO}\}$.

Now, we are able to define global type reduction, as follows.

Definition 5.7. We write $G \xrightarrow{\lambda} G'$ if $[\![G]\!] \xrightarrow{\lambda} [\![G']\!]$. In $G \xrightarrow{k^{\varphi}} G'$, we take off a prefix at k in G not suppressed by $\prec_{\mathsf{II}}$, $\prec_{\mathsf{IO}}$ or $\prec_{\mathsf{OO}}$, to obtain G'. In $G \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} G'$, we replace the topmost global type $\{\!|\tilde{k}^{\tilde{\varphi}}, \gamma_1, \gamma_2|\!\}$ with γ_2 to obtain G'.

5.3. *Subject reduction, communication safety and session fidelity*

In the following theorem we refer to structural rules defined in Figure 7.

Theorem 5.8 (subject congruence and reduction).

1. $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$ and $P \equiv P'$ imply $\Gamma \vdash P' \triangleright_{\tilde{r}} \Delta$.
2. $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$ such that Δ is coherent and $\Sigma \vdash P \to \Sigma' \vdash P'$ imply $\Gamma \vdash P' \triangleright_{\tilde{r}'} \Delta'$ where
   - ($\Delta = \Delta'$ and $\tilde{r} = \tilde{r}'$ and $\Sigma \subseteq \Sigma'$) or
   - ($\Delta \xrightarrow{s^{\varphi}} \Delta'$, for some s^{φ}, and $\tilde{r} = \tilde{r}'$ and $\Sigma = \Sigma'$) or
   - ($\Delta \xrightarrow{\mathsf{throw}(s^{\tilde{\varphi}})} \Delta'$, for some $\tilde{s^{\varphi}}$, and $\tilde{r} \subseteq \tilde{r}'$and $\Sigma = \Sigma'$).
3. $\Gamma \vdash P \triangleright_{\varnothing} \varnothing$ and $\Sigma \vdash P \to \Sigma' \vdash P'$ imply $\Gamma \vdash P' \triangleright_{\varnothing} \varnothing$.

The type discipline also satisfies, as in the preceding session type disciplines (Honda *et al.* 1998), communication error freedom, including linear usage of channels.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

A prefix is active if it is not under a prefix or under an if branch, after any unfoldings by (Def). Below we write $P\langle r!\rangle$ (resp. $P\langle r?\rangle$) if P contains an emitting (resp. receiving) active prefix at r, and we say that P has a *redex at r* if it has an active prefix at r among its redexes. The reduction context $\mathcal{E}$ is defined in Section 3.

Communication safety is standard except for the case in which there are pending receive actions that will not be reduced because the process containing the dual send has captured an exception and has been reduced to the handler. This is not an error since also the process containing the receive will be reduced to a compliant handler when capturing the same exception.

Theorem 5.9 (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) $P \equiv \mathcal{E}[s[\varphi] : L]$ such that either
   a. $P\langle s^{\varphi}?\rangle$, s^{φ} occurs exactly once in $\mathcal{E}$ and either $L \neq \varnothing$ or $P \equiv \mathcal{E}'[\mathsf{try}(\tilde{r}')\{R\}\ \mathsf{catch}\ \{Q'\} \mid s[\varphi+1] : L']$ for some $\mathcal{E}'$ with $s^{\varphi+1} \notin \tilde{r}'$ and $s^{\varphi} \in \tilde{r}'$ and $R\langle s^{\varphi}?\rangle$; or
   b. $P\langle s^{\varphi}!\rangle$ and s^{φ} occur exactly once in $\mathcal{E}$; or
   c. $P\langle s^{\varphi}?\rangle$, $P\langle s^{\varphi}!\rangle$, and s^{φ} occur 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
      i. either $P \equiv \mathcal{E}'[s[\varphi] : \tilde{v} \cdot L]$ for some $\mathcal{E}'$ and $|\tilde{v}| = |\tilde{y}|$.
      ii. or $P \equiv \mathcal{E}'[\mathsf{try}(\tilde{r}')\{R\}\ \mathsf{catch}\ \{Q'\} \mid s[\varphi+1] : L']$ for some $\mathcal{E}'$ with $s^{\varphi+1} \notin \tilde{r}'$ and $s^{\varphi} \in \tilde{r}'$.
   b. If $R \equiv s^{\varphi} \triangleright \{l_i : Q_i\}_{i\in I}$ then
      i. either $P \equiv \mathcal{E}'[s[\varphi] : l_j \cdot L]$ for some $\mathcal{E}'$ and $j \in I$.
      ii. or $P \equiv \mathcal{E}'[\mathsf{try}(\tilde{r}')\{R\}\ \mathsf{catch}\ \{Q'\} \mid s[\varphi+1] : L']$ for some $\mathcal{E}'$ with $s^{\varphi+1} \notin \tilde{r}'$ and $s^{\varphi} \in \tilde{r}'$.

As a corollary of the Theorem 5.8(2) we obtain *session fidelity*, which guarantees that the interactions of a typable process exactly follow the specification described by its global type.

Corollary 5.10 (session fidelity). Assume $\Gamma \vdash P \triangleright_{\tilde{t}} \Delta$ such that Δ is coherent and $\Delta(\tilde{s}) = [\![G]\!]$. If

— $\Sigma \vdash P\langle s_k^{\varphi}?\rangle \to \Sigma' \vdash P'$ at the redex of s_k^{φ}, then $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$, with either $G \xrightarrow{k} G'$ or $G \xrightarrow{\mathsf{throw}(\tilde{k})} G'$ and $k \in \tilde{k}$, and $[\![G']\!] = \Delta'(\tilde{s})$.
— $\Sigma \vdash P\langle s_k^{\varphi}!\rangle \to \Sigma' \vdash P'$ at the redex of s_k^{φ} then $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta$.

5.4. *Progress*

The type discipline ensures also the progress property, for a well-formed process. This notion of well formedness is given by the following definitions.

A process is queue-full when it has a queue for each session channel:

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Definition 5.11. Let $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$. Then P is *queue-full* when $\{\tilde{r}\}$ coincide with the set of session channels occurring in Δ.

A process is simple when each prefixed subterm in it has only a unique session.

Definition 5.12 (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 in Figure 9 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 5.13 (well linked). We say P is *well linked* when for each $\Sigma \vdash P \rightarrow^* \Sigma' \vdash Q$, whenever Q has an active prefix whose subject is a (free or bound) shared name, then it is always part of a redex.

The following proposition gives the converse of Corollary 5.10: if the global type has a reduction, then the process can always realise it. Notice that this is not true for throw-reduction because throws do not occur in a global protocol G. Then for every G of the form $\{\tilde{k}, \gamma_1, \gamma_2\}$ we can have $G \xrightarrow{\mathsf{throw}(\tilde{k})} G'$ but for the process to perform an analogous reduction step it needs to be a throw process explicitly inside it. This means that G is always ready to capture an exception if there is any, but exceptions are not part of the protocol since they are extemporaneous problematic situations and having those in the protocol would require each implementation to have a throw in the same exact point.

Proposition 5.14. Let $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$, Δ is coherent, P is simple, well linked and queue-full. Then:

1. If $P \not\equiv \mathbf{0}$ then $\Sigma \vdash P \rightarrow \Sigma' \vdash P'$ for some P'.
2. If $\Delta(\tilde{s}) = [\![G]\!]$ and $G \xrightarrow{k} G'$, then $\Sigma \vdash P \rightarrow^+ \Sigma' \vdash P'$ at the redex at s_k s. t. $\Gamma \vdash P' \triangleright_{\tilde{r}} \Delta'$ with $\Delta'(\tilde{s}) = [\![G']\!]$.

Noticing that a simple and queue-full program only reduces to a simple and queue-full program, we can state the following theorem:

Theorem 5.15 (progress). Let P be a simple and well-linked program. Then P has the *progress property* in the sense that $\Sigma \vdash P \rightarrow^* \Sigma' \vdash P'$ implies either $P' \equiv \mathbf{0}$ or $\Sigma \vdash P' \rightarrow \Sigma'' \vdash P''$ for some P'' and Σ''.

The following corollary of progress theorem gives us a *termination property*:

Corollary 5.16 (termination). Let $P\,;Q$ be a simple and well-linked program. Then it eventually reduces to Q, provided that P does not contain any infinite recursion. That is, either $\Sigma \vdash P\,;Q \rightarrow^* \Sigma \vdash Q$ or P is blocked by recursion.

5.5. *Throw confluence*

The following theorem states the confluence properties related to exceptions:

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

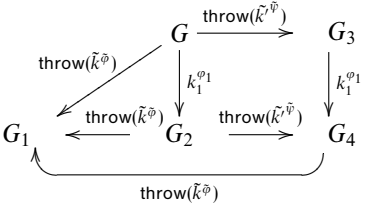
— If both an exception on a set of channels and a communication on a channel that is not affected by the exception can reduce at the same time, then the reduction order is not important, the process will end up in the same state.
— If both an exception and a communication on the same channel can reduce at the same time, then the order of reduction is not important, the process will end up in the same state.

The former is an atomicity property on exception, meaning that an exception does not implicitly affect external channels, in other words, an exception is not propagated to channels that were not explicitly mentioned in the related try-catch block.

For instance, consider the following global type:

$$G = \{\!|\tilde{k}^{\tilde{\varphi}},\ (\mathtt{p}_1 \to \mathtt{p}_2 : k_1^{\varphi_1}\langle \tilde{S}\rangle \mid \{\!|\tilde{k'}^{\tilde{\psi}},\ \gamma,\ \gamma'|\!\};\gamma_2),\ \gamma_3|\!\};\mathsf{end}$$

that describes a try-catch global behaviour, where $\tilde{k}^{\tilde{\varphi}}$ are the involved channels, $\mathtt{p}_1 \to \mathtt{p}_2 : k_1^{\varphi_1}\langle \tilde{S}\rangle \mid \{\!|\tilde{k'}^{\tilde{\psi}},\ \gamma,\ \gamma'|\!\};\gamma_2$ is the default behaviour of the enclosing try-catch block, while γ_3 describes the exception handling behaviour. We notice an enclosed try-catch block, namely $\{\!|\tilde{k'}^{\tilde{\psi}},\ \gamma,\ \gamma'|\!\}$, with $\tilde{k'} \subset \tilde{k}$ and $k_1 \notin \tilde{k'}$. Different observable reductions are possible for G, some of them are depicted in the diagram below.



Where

$$\begin{aligned}
G_1 &= \gamma_3;\mathsf{end},\\
G_2 &= \{\!|\tilde{k}^{\tilde{\varphi}},\ (\{\!|\tilde{k'}^{\tilde{\psi}},\ \gamma,\ \gamma'|\!\};\gamma_2),\ \gamma_3|\!\};\mathsf{end},\\
G_3 &= \{\!|\tilde{k}^{\tilde{\varphi}},\ (\mathtt{p}_1 \to \mathtt{p}_2 : k_1^{\varphi_1}\langle \tilde{S}\rangle \mid \gamma';\gamma_2),\ \gamma_3|\!\};\mathsf{end},\\
G_4 &= \{\!|\tilde{k}^{\tilde{\varphi}},\ (\gamma';\gamma_2),\ \gamma_3|\!\};\mathsf{end}.
\end{aligned}$$

If G reduces to G_1, G_2, or G_3, then the reductions $G_2 \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} G_1$, $G_2 \xrightarrow{\mathsf{throw}(\tilde{k'}^{\tilde{\psi}})} G_4$, and $G_4 \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} G_1$ are always possible. The theorem below generalizes these results.

Theorem 5.17 (throw confluence). If $G \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} G_1$ and $G \xrightarrow{k^{\varphi}} G_2$. Then:

1. if $k \notin \tilde{k}$ then $G_1 \xrightarrow{k^{\varphi}} G_3$ and $G_2 \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} G_3$.
2. if $k \in \tilde{k}$ then $G_2 \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} G_1$.

6. Coordinated exception handling and resolution: an example

6.1. *Coordinated atomic actions*

An atomic action consists in an interaction between a group of participants in which there are no interactions between the group and the rest of the system for the duration of the activity. Coordinated actions (CA) (Rubira and Wu 1995) are a generalization of

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

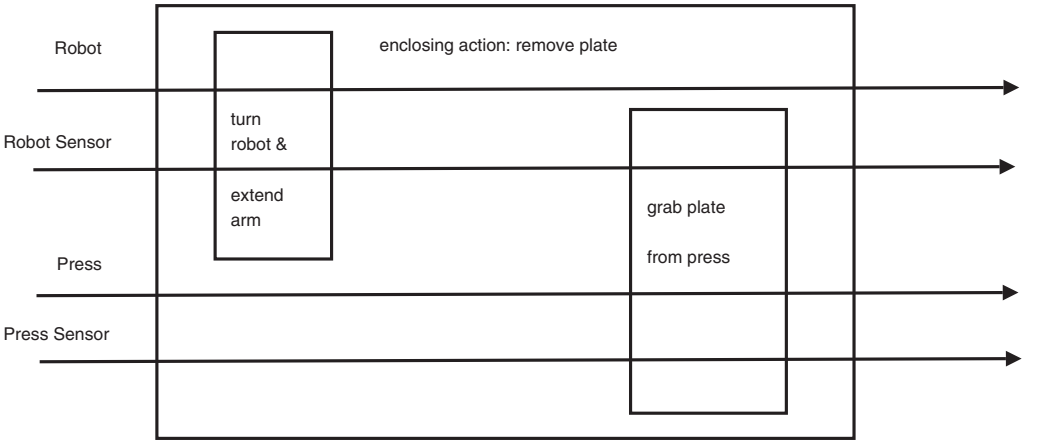


Fig. 12. The coordinated action `remove-plate`.

atomic actions and present a general technique for reaching fault tolerance in distributed OO systems by integrating conversations, transactions and exception handling. CA use conversations for controlling concurrency and communication between cooperating threads. During the execution of a CA, the access to shared *external objects* from other actions and threads should not be influenced by interactions between the CA action and the rest of the system. Rubira and Wu (1995) propose both a model and an algorithm for exception handling in CA actions. Exceptions can be raised by one or more cooperating threads: if something wrong happens normal execution is stopped and the control passes to the handlers. When a thread enters an action to play a specified role it also enters a specified associated exception contest: for each action various exceptions are defined and a resolution graph is declared in case several exception are raised concurrently. Disjoint subsets of participants may later enter in nested CA actions and consequently nested exception contexts (see Figure 12 illustrating an enclosing action and two nested actions. This example will be developed later). There are two kinds of exceptions:

— internal exceptions `E` which can be handled in the current action without alerting the enclosing action;
— exceptions **E** that must be signalled to the environment (the enclosing action or the whole system). These exceptions are raised when something happens that cannot be handled in the current action (during normal execution or exception handling: unrecoverable problem, delivering of incomplete results).

Exceptions can be propagated along chains of nested actions: if the local handling of an exception `E` is not successful, then a corresponding exception **E** 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) an algorithm determines which exception must be covered (solving possible conflicts in case of concurrent exceptions raised by cooperating threads).

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

We model the cooperating threads in a CA as a set of participants in a session; nested CAs are implemented by try-catch blocks involving only a subset of participants/channels: participants can interact in the default processes P's and in case of failure/problem they can move to the handlers. For each (nested) CA we assume an 'exception resolver'. This participant is inactive during normal execution then, when one or more exceptions are thrown, it collects the messages concerning the kind of exceptions that have been raised by one or more participants: in case more than one exception has been raised in the same action it decides which one has the priority (according to the resolution graph) and then it sends the corresponding label to all participants.

Let $\{\mathrm{E}_h\}_{h\in H}$ and $\{\mathbf{E}_k\}_{k\in K}$ be sets of internal and external exceptions respectively and

$$(\nu\tilde{s}')\begin{pmatrix} \cdots \mid \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{\mathbf{0}\}\ \mathsf{catch}\ \{Q_1\}\}\mathsf{catch}\{Q_1'\} \mid \\ \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{P_2\}\ \mathsf{catch}\ \{Q_2\}\}\mathsf{catch}\{Q_2'\} \mid \\ \cdots \mid \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{P_n\}\ \mathsf{catch}\ \{Q_n\}\}\mathsf{catch}\{Q_n'\} \mid \\ s_1[0] : \varnothing \mid \cdots \mid s_m[0] : \varnothing \end{pmatrix}$$

represent a CA where $\tilde{s} \subset \tilde{s}' = \{s_1, \ldots s_m\}$ and $\mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{\mathbf{0}\}\ \mathsf{catch}\ \{Q_1\}\}\mathsf{catch}\{Q_1'\}$ is the resolver process. $\tilde{s}'$ 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:

$$Q_1 = \mathsf{try}(\tilde{s})\{s_1^1?(x_2).\ldots s_n^1?(x_n).;\ \mathsf{selects}\ h \in H.\ s_n^1!\langle h\rangle \ldots s_1^1!\langle h\rangle\}\ \mathsf{catch}\ \{\mathbf{0}\} \quad (1)$$

Q_1 in its default part collects from all participants the labels denoting the kind of exceptions that have been raised. Then it selects in the exception graph the one that has to be resolved and sends to participants the corresponding label. Dually, the handlers Q_j for $j \in \{2, \ldots, n\}$ have the shape:

$$Q_j = \mathsf{try}(\tilde{s})\{\mathsf{if}\ \mathtt{test} = h\ \mathsf{then}\ s_i^1!\langle h\rangle\ \mathsf{else}\ \ s_i^1!\langle 0\rangle; s_j^1?(x).; Q_j'\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}')\} \quad (2)$$

where `test` checks whether exception h has been raised by the current participant. If $\mathtt{p}_j$ has not raised exception it sends 0. Then Q_j' starts handling the exception. Let us notice that the handler of the current block is a throw on the sets of channels involved in the enclosing action. The reason is that if something goes wrong handling the current exception (that is $Q_j' \xrightarrow{*} \mathsf{throw}(\tilde{s}^1)$) the execution passes to the enclosing action by throwing and exception on $\tilde{s}'$.

6.2. *Production cell*

We now focus on 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 (Lewerentz and Lindner 1995). 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 Xu *et al.* (1998). Here,

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

we just model the part of the system responsible for removing the plate from the press. The components we are considering are `Robot`, `RobotSensor`, `Press` and `PressSensor` abbreviated respectively as `R`, `RS`, `P` and `PS` (see Figure 12). 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 `RF` (robot failure), `RSF` (robot sensor failure), `PF` (press failure), and `PSF` (press sensor failure). Processes are indexed by participants' abbreviations for instance $Q_{\mathtt{RS}}$ 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` and `BPS`.

The enclosing action `remove-plate` uses channels $\tilde{s}'$. Concerning nested actions channel $\tilde{s}$ is used in action `turn-robot-and-extend-arm` while channel $\tilde{s}''$ is used in action `grab-plate-from-press`. We recall that we write s as a shorthand for s^0. The action `remove-plate` can be encoded as the following session:

$$\varnothing \vdash (\nu \tilde{s}')(\mathtt{ResolverTR} \mid \mathtt{Robot} \mid \mathtt{RobotSensor} \mid \mathtt{Press} \mid \mathtt{PressSensor} \mid \mathtt{ResolverGP} \mid s_1[0] : L_1 \mid \ldots \mid s_n[0] : L_n)$$

where

$$\begin{aligned}
\mathtt{ResolverTR} &= \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{\mathbf{0}\} \ \mathsf{catch} \ \{R_{\mathtt{TR}}\}\} \ \mathsf{catch} \ \{\mathbf{0}\}\\
\mathtt{ResolverGP} &= \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{\mathbf{0}\} \ \mathsf{catch} \ \{R_{\mathtt{GP}}\}\} \ \mathsf{catch} \ \{\mathbf{0}\}\\
\mathtt{Robot} &= \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{P_R\} \ \mathsf{catch} \ \{Q_{\mathtt{R}}\}\} \ \mathsf{catch} \ \{Q'_{\mathtt{R}}\}\\
\mathtt{RobotSensor} &= \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{P_{\mathtt{RS}}\} \ \mathsf{catch} \ \{Q_{\mathtt{RS}}\}; \mathsf{try}(\tilde{s}'')\{P'_{\mathtt{RS}}\} \ \mathsf{catch} \ \{Q'_{\mathtt{RS}}\}\} \ \mathsf{catch} \ \{Q''_{\mathtt{RS}}\}\\
\mathtt{Press} &= \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}'')\{P_{\mathtt{P}}\} \ \mathsf{catch} \ \{Q_{\mathtt{P}}\}\} \ \mathsf{catch} \ \{Q'_{\mathtt{P}}\}\\
\mathtt{PressSensor} &= \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}'')\{P_S\} \ \mathsf{catch} \ \{Q_{\mathtt{PS}}\}\} \ \mathsf{catch} \ \{Q'_{\mathtt{PS}}\}.
\end{aligned}$$

Let us notice that each nested action has a corresponding resolver. The semantics of this example can be found in Appendix B.

6.3. *Correctness and complexity*

Let $s' = \{s_{i_1} \ldots s_{i_n}\}$, $s = \{s_{j_1} \ldots s_{j_m}\}$ and $s'' = \{s_{h_1} \ldots s_{h_\ell}\}$. The global type of the production cell example is:

$$\{(i_1, \ldots, i_n), \{(j_1, \ldots, j_m), \gamma_{\mathtt{R/RS}}, \gamma'_{\mathtt{R/RS}}\}; \{(h_1, \ldots, h_\ell), \gamma_{\mathtt{R/PS/P}}, \gamma'_{\mathtt{R/PS/P}}\}, \gamma'\}$$

where $\gamma_{\mathtt{R/RS}}$, $\gamma'_{\mathtt{R/RS}}$ describe respectively the default and handler interactions between `Robot` and `RobotSensor` in `turn-robot-and-extend-arm` action, while $\gamma_{\mathtt{R/RS/P}}$, $\gamma'_{\mathtt{R/RS/P}}$ describe respectively the default and handler interactions between `Robot Sensor`, `Press` and `PressSensor` in `grab-plate-from-press` action. Finally γ' describes the protocol of the handler related to the enclosing action `remove-plate`.

In Xu *et al.* (1998), some assumptions are made in order to prove correctness and complexity results. These assumptions are all satisfied by well-typed protocols in our model:

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

— no message loss or corruption: at protocol level the correctness of messages send-receive is guaranteed by communication safety (Theorem 5.9);
— FIFO message passing: two messages from thread $\mathtt{p}_i$ will arrive at thread $\mathtt{p}_j$ in the same order as they were sent; this is ensured by session fidelity (Corollary 5.10);
— absence of deadlock during exception handling. This is ensured by progress (Theorem 5.15).

Let N be 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} be 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 Xu *et al.* (1998) the following results:

1. Any participant $\mathtt{p}$, 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 plus N throws so the cost depends on the cost of throw implementation. Let us notice that in Xu *et al.* (1998), the algorithm performs in $\mathcal{O}(N^2)$ messages.

Proof of result 1. Let us consider the worst case, i.e. a thread that raises an exception is in the innermost CA action and each time the abortion of a nested action occurs right at the end of exception handling within that nested action. Assume that a thread $\mathtt{p}_i$ in the innermost action raises an exception. It will send the exception message to the corresponding resolver and put a throw in Σ. Since there are no further nested actions within the innermost action, any message to/from the other threads about an exception will just come from/to the resolver thread of the current action in at most $2T_{nmax}$. Actual exception resolution may take T_{reso}. Therefore, $\mathtt{p}_i$ will receive a resolving exception and then complete exception handling in at most $3T_{nmax} + T_{reso} + \Delta_{nmax} + T_{abort} + T_{throw}$.

If the current action is not the innermost one, and an exception occurs in its direct containing action, then the innermost action has to be aborted. After the abortion (which costs T_{abort}), $\mathtt{p}_i$ will then receive the exception by the resolver in at most $T_{nmax} + T_{reso}$ and complete exception handling within Δ_{nmax}. The whole process costs at most $T_{nmax} + T_{abort} + T_{reso} + \Delta_{nmax}$. In the worst case, the above process could be repeated $nmax$ times until the outermost CA action is reached. Totally the repeated process will cost at most $nmax(T_{nmax} + T_{abort} + T_{reso} + \Delta_{nmax})$. Adding the time spent in the innermost action, we therefore have that

$$T \leqslant (nmax + 3)T_{nmax} + nmax\,T_{abort} + (nmax + 1)(T_{reso} + \Delta_{nmax}) + T_{throw}$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

namely, thread p_i will complete exception handling ultimately and leave the outermost CA action.

Proof of result 2. Assume that a new exception message arrives at the resolving thread after it has started the resolution. Note that, the resolver thread must receive a label from each participating thread in A before it can begin any actual resolution. Hence the only possibility is that the newly arriving exception is caused by an abortion event, namely, A must be aborted by some enclosing action, contradicting the assumption that no exception is raised in any enclosing action of A.

Proof of result 3. An exception that is raised in the containing CA action aborts the nested action (even if a resolving exception for the nested action has been found). Note that however the number of nesting levels is finite and bounded by *nmax*. Abortion will be no longer possible if the current active action A is the outermost (or top-level) CA action. By Lemma 2, the exception resolution will start finally and no more new exception will be raised.

Proof of result 4. Without the nesting of CA actions, the message complexity of our approach is $\mathcal{O}(N)$ messages, where N is the number of the threads participating in the outermost CA action. More precisely: (1) when only one exception is raised and there are no nested actions, then the number of messages is the cost of a throw that can be compared to one message, $(N-1)$ messages to the resolver, and $(N-1)$ commit messages; (2) when all N participating threads have the exceptions raised simultaneously, the number of messages is $(N-1)$ throws, $2(N-1)$ messages to/from the resolver and $(N-1)$ commit messages.

7. Related work

In the context of session types theory, (Carbone *et al.* 2008) and(Carbone 2009) proposed *interactional* exceptions, which inspired our work, for binary sessions and for web service choreographies. The approach described in Carbone *et al.* (2008) and Carbone (2009) 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; (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.

Exception handling mechanisms have been studied for many programming languages including communication based ones: in distributed object-oriented programming (Rubira and Wu 1995; Xu *et al.* 1998), in particular Xu *et al.* (1998) presents the algorithm we implemented as an example in Section 6. CAs have been adapted in Tartanoglu *et al.* (2003)

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

to model fault tolerant web services: the resulting web service composition actions 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 do not address this issue in the present article.

Jakšić and Padovani (2012) presents a copyless message passing model with exceptions and delegation where communicating processes share a common heap. The system keeps a track of the resources allocated during executions and it is restored to a consistent configuration in the case an exception is thrown. Since they do not focus on distributed systems, they relax our constraints in modelling nested exceptions: nested exceptions can involve an unrestricted number of channels and outer exception handlers can take control over inner blocks at any point of execution.

Several service-oriented calculi (e.g. Bocchi *et al.* (2003), Lapadula *et al.* (2007) and Vieira *et al.* (2008)) 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 Calculus of Sessions and Pipelines (CaSPiS Boreale *et al.* (2008)) is a process calculus for web services modelling binary (possibly nested) sessions equipped with a pipeline mechanism for communications between inner and outer sessions. In CaSPiS, a session can be explicitly terminated by any of the two sides. Both communicating peers have an associated termination handler which is activated when the other peer sends a session termination signal. Similarly, to our approach in CaSPIs all nested sessions are terminated when the outer one is closed. On the other hand, our model is more complex both because of multiparty sessions and because of exceptions structure. In CaSPIs any side terminating the session can start killing nested subsessions; instead, in our model, all participants involved in an inner exception are required to catch it before letting the outer one to abort the inner blocks.

Bravetti and Zavattaro (2009) investigates the decidability of termination problems for two simple fragments of CCS (one with recursion and one with replication) extended with the try-catch operator for exception handling. Also the *interrupt* operator of CSP is considered, but this has a different semantics, and does not compare with our approach. They are interested in expressiveness results and not in enforcing correctness of the communication. The interest of this work with respect to ours is in that it provides a theoretical account on the interplay between exception handling and recursion, featured also by our calculus. They prove that termination is undecidable when recursion and try-catch block coexist.

The paper of Lanese *et al.* (2010) 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.

8. Conclusions

We have introduced a type-safe global escape mechanism for handling unexpected or unwanted conditions changing the default execution of distributed communication flows, by means of a collection of asynchronous local exceptions. All the involved conversation

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

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 (Honda *et al.* 2008), and showed 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. Actually our model covers only those kind of errors that do not compromise the system robustness: for instance we do not handle hardware failures and we always assume that there are no failures in message passing.

The flexibility was achieved 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 theclient aborts the interaction or the client accepts the wrong loan offer from the bank.

We tackled many aspects that have to be considered when designing a model for exception handling in distributed interactions (Lanese and Montesi 2010):

— *full specification,* our model defines behaviour for all possible cases including the handling of concurrent exceptions,
— *intuitiveness,* we use intuitive constructs (try-catch blocks and throw), well known to programmers which facilitate the design of error handling scenarios,
— *minimality,* we add just two constructs, one for try-catch blocks and one for exception raising thus keeping the simplicity and intuitiveness of the original multiparty model.

Our type discipline has been proved to ensure all the main standard properties, such as *subject reduction* (Theorem 5.8), *communication error freedom* (Theorem 5.9), *fidelity of the interactions w.r.t. the global protocol* (Corollary 5.10), and the *progress property* (Theorem 5.15). Moreover, we stated and prove two further results: a form of *termination property* (Corollary 5.16), and, finally, the *modularity* of exception handling (Theorem 5.17).

We obtained the above results by means of:

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 peer behaviours.

8.1. *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.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

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). A more advanced topic is a combination with nested subsessions (Demangeon and Honda 2012) and asynchronous exceptions where one could escape the scopes from a child session to its parent session without destructing the session structures. Finally, we plan to integrate a rollback mechanism allowing the recovery of part of the consumed session: this mechanism should allow programmers to indicate specific return points to which the interaction should come back in case of exception.

A variant of our exception mechanism is already included as the construct `interrupt` into Scribble 1.0 (Honda *et al.* 2011, 2012), a language to describe application-level protocols among communicating systems based on the multiparty session type theory with runtime monitoring mechanisms (Hu *et al.* 2013; Neykova *et al.* 2013). A full integration of type-checking with session-based end-point languages such as Java (Hu *et al.* 2008, 2010) is an ongoing work.

Acknowledgements

The initial idea of global escape and a part of the examples in this article are originated by the first Scribble document by Kohei Honda.

Appendix A. The Client-Seller-Bank Example

A.1. *Semantics*

In this section, we show the semantics of our client-seller-bank protocol. First we recall the syntax by adding details about C-B interaction (Figure A1). B 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 ($\mathsf{let}\ h = \mathtt{timer}(0)\ \ \mathsf{in}\ \ldots$). Thus there are two iterations in the process: one related to the deal ($\mathsf{def}\ X$) and the other used by B to iterate on time intervals ($\mathsf{def}\ Y$). C examines B offer and, if she agrees on it, she throws an exception to exit the iteration (this is implemented as an inner try-catch block involving only s_3, s_4: when the throw is raised, S is not involved) otherwise, she waits for another offer and iterates the negotiation. Dually B 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. B calculates another offer then iterates the outer recursive process sending the new loan to C. An interesting scenario is when the time-out and the acceptance of the loan from C rise concurrently. It can then happen that C has accepted an offer while B was updating his offer after a time-out: B and C agreed on different amounts of money. This is resolved by the handlers: after the exception has been thrown B sends to C the latest value of the loan. C checks it and decide whether to accept it or not. In the latter case, she sends a NOK label to B and then aborts the transaction by throwing an exception.

The time intervals are modelled through a *timer* construct whose semantics is:

$$\mathsf{let}\ h = \mathtt{timer}(t)\ \ \mathsf{in}\ P \longrightarrow\ \mathsf{let}\ h = \mathtt{timer}(t+1)\ \ \mathsf{in}\ P\ \ [\lfloor \textsc{Timer} \rfloor]$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$
\begin{array}{lcl}
\mathtt{S} & = & \mathtt{BS}[1](s_1,s_2,s_3,s_4).\mathsf{try}(s_1,s_2,s_3,s_4)\{s_1?(o).s_2!\langle \mathtt{a}\rangle; s_1?(m).c = \mathtt{true}; s_2!\langle \mathtt{d}\rangle\} \\
& & \quad \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathsf{if}\ (c)\ \mathsf{then}\ \{s_2^1!\langle \mathtt{d}\rangle\} \\
& & \quad\quad \mathsf{else}\ \mathsf{throw}(s_1^1,s_2^1,s_3^2,s_4^2)\} \\
& & \mathsf{catch}\{\mathtt{abort}\}\} \\
\\
\mathtt{C} & = & \overline{\mathtt{BS}}[2](s_1,s_2,s_3,s_4).\mathsf{try}(s_1,s_2,s_3,s_4)\{s_1!\langle \mathtt{o}\rangle; s_2?(a); P'; s_1!\langle \mathtt{m}\rangle; (s_2?(d) \mid \mathsf{throw}(s_1,s_2,s_3,s_4))\} \\
& & \quad \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{s_2^1?(d)\}\mathsf{catch}\{\mathtt{abort}\}\} \\
\\
P' & = & s_3!\langle \mathtt{co}\rangle; \\
& & \mathsf{try}(s_3,s_4)\{s_4 \rhd \{\mathtt{OK} : \mathbf{0}, \mathtt{NEM} : \mathsf{def}\ X(s_3,s_4) = s_4?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ \mathsf{throw}(s_3,s_4)\ \mathsf{else}\ X\langle s_3,s_4\rangle \\
& & \quad\quad \mathsf{in}\ X\langle s_3,s_4\rangle\} \\
& & \mathsf{catch}\{s_3^1!\langle \mathtt{f}\rangle; s_4^1 \rhd \{\mathtt{OK} : \mathbf{0}, \mathtt{NOK} : \mathsf{throw}(s_1,s_2,s_3^1,s_4^1))\}\}\} \\
\\
\mathtt{B} & = & \overline{\mathtt{BS}}[3](s_1,s_2,s_3,s_4).\mathsf{try}(s_1,s_2,s_3,s_4)\{P''\}\mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\} \\
\\
P'' & = & s_3?(co).\quad \mathsf{try}(s_3,s_4)\{\mathsf{if}\ \mathit{enoughmoney}\ \mathsf{then}\ s_4 \lhd \mathtt{OK}.\mathbf{0} \\
& & \quad\quad \mathsf{else}\ s_4 \lhd \mathtt{NEM}.\mathsf{def}\ X(s_3,s_4) = \ \mathsf{let}\ h = \mathtt{timer}(0)\ \ \mathsf{in} \\
& & \quad\quad\quad \mathsf{def}\ Y(s_4) = s_4!\langle \mathtt{f}\rangle; \mathsf{if}\ h = \mathit{timeout}\ \mathsf{then}\ \mathrm{calculate}\{\mathtt{f}\}; X\langle s_3,s_4\rangle \\
& & \quad\quad\quad\quad\quad \mathsf{else}\ Y\langle s_4\rangle\ \ \mathsf{in}\ Y\langle s_4\rangle\ \mathsf{in}\ X\langle s_3,s_4\rangle\} \\
& & \quad\quad \mathsf{catch}\{s_3^1?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ s_4^1 \lhd \mathtt{OK}.\mathbf{0}\ \mathsf{else}\ \ s_4^1 \lhd \mathtt{NOK}.\mathbf{0}\}
\end{array}
$$

Fig. A1. Criss-cross example (full code of Example 1 with properly levelled channels).

For instance, let us suppose B updates the offer every 2 time-intervals; after three reductions ($\lfloor$Timer$\rfloor$ rule) the if-test succeeds:

$$\mathsf{def}\ X(s_3,s_4) = \ \mathsf{let}\ h = \mathtt{timer}(0)\ \ \mathsf{in}\ \ \mathsf{def}\ Y(s_4)\mathsf{if}\ h = \mathit{timeout}\ \mathsf{then}\ \mathrm{calculate}\{\mathtt{f}\};$$

$$s_4!\langle \mathtt{f}\rangle; X\langle s_3,s_4\rangle\ \mathsf{else}\ \ Y\langle s_4\rangle \longrightarrow$$

$$\mathsf{def}\ X(s_3,s_4) = \ \mathsf{let}\ h = \mathtt{timer}(1)\ \ \mathsf{in}\ \ \mathsf{def}\ Y(s_4) = \mathrm{calculate}\{\mathtt{f}\}; s_4!\langle \mathtt{f}\rangle; X\langle s_3,s_4\rangle$$

Now let us analyse in details the main interesting interactions. We start from:

$$\varnothing \vdash \mathtt{S} \mid \mathtt{C} \mid \mathtt{B}$$

and after the application of the rule (Link) we obtain:

$$
\begin{array}{rl}
\varnothing \vdash (\nu s_1,s_2,s_3,s_4) & \mathsf{try}(s_1,s_2,s_3,s_4)\{s_1?(o).s_2!\langle \mathtt{a}\rangle; s_1?(m).c = \mathtt{true}; s_2!\langle \mathtt{d}\rangle\} \\
& \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathsf{if}\ (c)\ \mathsf{then}\ \{s_2^1!\langle \mathtt{d}\rangle\}\mathsf{else}\ \mathsf{throw}(s_1^1,s_2^1,s_3^2,s_4^2)\} \\
& \quad \mathsf{catch}\{\mathtt{abort}\}\} \mid \\
\\
& \mathsf{try}(s_1,s_2,s_3,s_4)\{s_1!\langle \mathtt{o}\rangle; s_2?(a); P'; s_1!\langle \mathtt{m}\rangle; (s_2?(d) \mid \mathsf{throw}(s_1,s_2,s_3,s_4))\} \\
& \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{s_2^1?(d)\}\mathsf{catch}\{\mathtt{abort}\}\} \mid \\
& \mathsf{try}(s_1,s_2,s_3,s_4)\{P''\}\mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\} \mid \\
& s_1[0] : \varnothing \mid s_2[0] : \varnothing \mid s_3[0] : \varnothing \mid s_4[0] : \varnothing).
\end{array}
$$

Let us suppose S sends a confirmation to C and concurrently C throws an exception because she has waited too long. After the send/receive of the order and the send of the

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

confirmation we apply rule ⌊Thr⌋:

$$\begin{array}{l}\mathsf{throw}(s_1,s_2,s_3,s_4) \vdash (\nu s_1,s_2,s_3,s_4)\\ \qquad \mathsf{try}(s_1,s_2,s_3,s_4)\{\mathbf{0}\}\\ \qquad \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathsf{if}\ (c)\ \mathsf{then}\ \{s_2^1!\langle \mathtt{d}\rangle\}\mathsf{else}\ \mathsf{throw}(s_1^1,s_2^1,s_3^2,s_4^2)\}\\ \qquad\qquad \mathsf{catch}\{\mathtt{abort}\}\}\ |\\ \\ \qquad \mathsf{try}(s_1,s_2,s_3,s_4)\{s_2?(a);P';s_1!\langle \mathtt{m}\rangle;s_2?(d)\}\\ \qquad \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{s_2^1?(d)\}\mathsf{catch}\{\mathtt{abort}\}\}\ |\\ \\ \qquad \mathsf{try}(s_1,s_2,s_3,s_4)\{P''\}\mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\}\ |\\ \qquad s_1[0]:\varnothing\ |\ s_2[0]:\mathtt{a}\ |\ s_3[0]:\varnothing\ |\ s_4[0]:\varnothing).\end{array}$$

Then rule ⌊RThr⌋ is applied twice and both `C` and `S` pass the control to the handlers. Since the confirmation has been sent and $c = \mathsf{true}$ `S` sends the date again and the transaction is successfully completed:

$$\begin{array}{l}\mathsf{throw}(s_1,s_2,s_3,s_4) \vdash (\nu s_1,s_2,s_3,s_4)(\ \mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{s_2^1!\langle \mathtt{d}\rangle\}\mathsf{catch}\{\mathtt{abort}\}\ |\\ \qquad\qquad \mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{s_2^1?(d)\}\mathsf{catch}\{\mathtt{abort}\}\ |\\ \qquad\qquad \mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\ |\\ \qquad\qquad \ldots)\end{array}$$

We see here the importance of the premise on channel levels in rule ⌊Thr⌋: the throw is reduced only if the levels of its argument channels are greater or equal then the level in the current try . If there is at least one level in the throw which is strictly minor then the one in the try then the throw is 'old' and should not be reduced. In the above case, the premise forbid the reduction of $\mathsf{throw}(s_1,s_2,s_3,s_4)$ in Σ which would abort a successful transaction.

Now let us focus on another possible escape in the interaction. Let us come back to `C-B` negotiation:

$$\begin{array}{l}\varnothing \vdash (\nu s_1,s_2,s_3,s_4)\ldots\\ \\ \qquad \mathsf{try}(s_1,s_2,s_3,s_4)\{P';s_1!\langle \mathtt{m}\rangle;s_2?(d)\}\\ \qquad \mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{s_2^1?(d)\}\mathsf{catch}\{\mathtt{abort}\}\}\ |\\ \\ \qquad \mathsf{try}(s_1,s_2,s_3,s_4)\{P''\}\mathsf{catch}\{\mathsf{try}(s_1^1,s_2^1,s_3^2,s_4^2)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\}\ |\\ \qquad \ldots\end{array}$$

Let us concentrate on the interaction $P' \mid P''$:

$$\begin{array}{l}\varnothing \vdash \begin{array}{l}s_3!\langle \mathtt{co}\rangle;\\ \mathsf{try}(s_3,s_4)\{s_4 \rhd \{\mathtt{OK}:\mathbf{0},\mathtt{NEM}:\mathsf{def}\ X(s_3,s_4)=s_4?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ \mathsf{throw}(s_3,s_4)\ \mathsf{else}\\ \qquad\qquad X\langle s_3,s_4\rangle\mathsf{in}\ X\langle s_3,s_4\rangle\}\\ \mathsf{catch}\{s_3^1!\langle \mathtt{f}\rangle;s_4^1 \rhd \{\mathtt{OK}:\mathbf{0},\mathtt{NOK}:\mathsf{throw}(s_1,s_2,s_3^1,s_4^1))\}\}\}\ |\end{array}\end{array}$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$s_3?(co).\ \mathsf{try}(s_3,s_4)\{\mathsf{if}\ \mathit{enoughmoney}\ \mathsf{then}\ s_4 \lhd \mathtt{OK}.\mathbf{0}$$
$$\mathsf{else}\ s_4 \lhd \mathtt{NEM}.\mathsf{def}\ X(s_3,s_4) = \ \mathsf{let}\ h = \mathtt{timer}(0)\ \ \mathsf{in}$$
$$\mathsf{def}\ Y(s_4) = s_4!\langle \mathtt{f}\rangle;\mathsf{if}\ h = \mathit{timeout}\ \mathsf{then}\ \mathsf{calculate}\{\mathtt{f}\};X\langle s_3,s_4\rangle$$
$$\mathsf{else}\ Y\langle s_4\rangle\ \ \mathsf{in}\ Y\langle s_4\rangle\ \mathsf{in}\ X\langle s_3,s_4\rangle\}$$
$$\mathsf{catch}\{s_3^1?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ s_4^1 \lhd \mathtt{OK}.\mathbf{0}\ \mathsf{else}\ \ s_4^1 \lhd \mathtt{NOK}.\mathbf{0}\}.$$

After C has sent the code of his account to B there are two cases: C has enough money or not. The most interesting case is the second one. In this case, B takes the second branch of the condition and sends the label NEM to C. After the send/receive of the label we have:

$$\varnothing \vdash \begin{array}{l}\mathsf{try}(s_3,s_4)\{\mathsf{def}\ X(s_3,s_4) = s_4?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ \mathsf{throw}(s_3,s_4)\ \mathsf{else}\ \ X\langle s_3,s_4\rangle\\ \qquad\qquad \mathsf{in}\ X\langle s_3,s_4\rangle\}\\ \mathsf{catch}\{s_3^1!\langle \mathtt{f}\rangle;s_4^1 \rhd \{\mathtt{OK} : \mathbf{0}, \mathtt{NOK} : \mathsf{throw}(s_1,s_2,s_3^1,s_4^1))\}\}\ |\end{array}$$

$$\begin{array}{l}\mathsf{try}(s_3,s_4)\{\mathsf{def}\ X(s_3,s_4) = \ \mathsf{let}\ h = \mathtt{timer}(0)\ \ \mathsf{in}\\ \quad \mathsf{def}\ Y(s_4) = s_4!\langle \mathtt{f}\rangle;\mathsf{if}\ h = \mathit{timeout}\ \mathsf{then}\ \mathsf{calculate}\{\mathtt{f}\};X\langle s_3,s_4\rangle\\ \qquad\qquad \mathsf{else}\ Y\langle s_4\rangle\ \ \mathsf{in}\ Y\langle s_4\rangle\ \mathsf{in}\ X\langle s_3,s_4\rangle\}\\ \mathsf{catch}\{s_3^1?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ s_4^1 \lhd \mathtt{OK}.\mathbf{0}\ \mathsf{else}\ \ s_4^1 \lhd \mathtt{NOK}.\mathbf{0}\}\end{array}$$

Now let us suppose timeout and rate acceptance are reached concurrently, so first B send the new fare f to C then C throws the exception $\mathsf{throw}(s_3,s_4)$ which is put into Σ then applying rule $\lfloor$RThr$\rfloor$ and coming back to the complete interaction we obtain:

$$\begin{array}{l}\mathsf{throw}(s_3,s_4) \vdash (\nu s_1,s_2,s_3,s_4)(\ldots\ |\\ \quad \mathsf{try}(s_1,s_2,s_3,s_4)\{s_3^1!\langle \mathtt{f}\rangle;s_4^1 \rhd \{\mathtt{OK} : \mathbf{0}, \mathtt{NOK} : \mathsf{throw}(s_1,s_2,s_3^1,s_4^1))\}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\ |\\ \quad \mathsf{try}(s_1,s_2,s_3,s_4)\{s_3^1?(f).\mathsf{if}\ \mathtt{OK}(f)\ \mathsf{then}\ s_4^1 \lhd \mathtt{OK}.\mathbf{0}\ \mathsf{else}\ \ s_4^1 \lhd \mathtt{NOK}.\mathbf{0}\}\ \mathsf{catch}\ \{\mathtt{abort}\}\\ \quad |\ \ldots\ |\ s_3[1] : \varnothing\ |\ s_4[1] : \mathtt{f}).\end{array}$$

Let us notice that our implementation of the protocol never moves to the situation where:

— S sends a confirmation to C but C aborts the interaction.
— C accepts the wrong loan offer from B.

A.2. *Global type*

In Figure A2, we present the complete global type of the client-seller-bank example. The whole interaction is part of a try-catch block involving all channels (1, 2, 3, 4). The global type of the default interaction starts at line 1 and ends at line 11. If the default protocol fails the transaction is aborted, for this reason the global type of the handler in line 12 is ϵ which denotes the empty communication. Line 1 says that C sends a string (the order) to S using channel 1. Line 2 says that S sends a string (the confirmation) to C using channel 2. In line 3 C sends to B a string (the loan proposal) using channel 3, then in line 4 we find a nested try block involving channels 3 and 4. This nested block describes the interaction between the client and the bank (lines 4–7 for the default interaction, lines 8–9 for the handler). Lines 4–5 say that B sends to C on channel 4 one of the labels OK or NEM. In case OK is sent there are no more communications and the interaction goes on from line

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

$$
\begin{array}{ll}
1. & \{(1,2,3,4), \quad \mathtt{C} \to \mathtt{S} : 1\langle \mathsf{string} \rangle; \\
2. & \qquad\quad \mathtt{S} \to \mathtt{C} : 2\langle \mathsf{string} \rangle; \\
3. & \qquad\quad \mathtt{C} \to \mathtt{B} : 3\langle \mathsf{string} \rangle; \\
4. & \qquad\quad \{(3,4), \mathtt{B} \to \mathtt{C} : 4\{\mathtt{OK} : \epsilon, \\
5. & \qquad\qquad\qquad\qquad \mathtt{NEM} : \mu \mathbf{t}.\mathtt{B} \to \mathtt{C} : 4\langle \mathsf{double} \rangle; \\
6. & \qquad\qquad\qquad\qquad\qquad \mathtt{C} \to \mathtt{B} : 3\{\mathtt{OK} : \epsilon, \\
7. & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{NOK} : \mathbf{t}\}, \\
8. & \qquad\qquad\quad \mathtt{B} \to \mathtt{C} : 4\langle \mathsf{double} \rangle; \\
9. & \qquad\qquad\quad \mathtt{C} \to \mathtt{B} : 3\{\mathtt{OK} : \epsilon, \mathtt{NOK} : \epsilon\}\}, \\
10. & \qquad\quad \mathtt{C} \to \mathtt{S} : 1\langle \mathsf{money} \rangle; \\
11. & \qquad\quad \mathtt{S} \to \mathtt{C} : 2\langle \mathsf{date} \rangle, \\
12. & \qquad \epsilon\}
\end{array}
$$

Fig. A2. The complete global type of the client-seller-bank example.

10 after the end of the current try-catch block. Instead if NEM is sent (line 5) a negotiation starts at line 5 described by the recursive type $\mu \mathbf{t}$: first B sends a double to C on channel 4 then it waits for reply from C at channel 3. In case OK is sent (line 6) the negotiation can be stopped (by a throw which is not represented by any type as explained in Section 1.1) and the protocol can go on with the handler (lines 8–9). In case NOK is sent (line 7) the communication is iterated using type $\mathbf{t}$. In the handler (lines 8–9) B sends a double to C on channel 4 to confirm the loan (line 8) then C sends to B at line 9 either OK (in this case the interaction goes on from line 10) or NOK (in this case an exception is raised which abort the transaction). Finally, line 10–11 says that C sends money to S using channel 1 and that S sends a date to C using channel 2.

Appendix B. The Semantics of Production Cell Example

In this section, we show the semantics of the production cell example. We analyse possible failure cases in nested actions and their handling.

Let us focus on Robot | RobotSensor and suppose there is a failure in the Robot and concurrently in the Robot-sensor, that is $P_{\mathtt{R}} = \mathsf{throw}(\tilde{s}) \mid P'_{\mathtt{R}}$ and $P_{\mathtt{RS}} = \mathsf{throw}(\tilde{s}) \mid P'_{\mathtt{RS}}$:

$$
\begin{array}{l}
\varnothing \vdash \mathtt{ResolverTR} \mid \mathtt{Robot} \mid \mathtt{RobotSensor} \longrightarrow^{*} \\
\varnothing \vdash \mathtt{ResolverTR} \mid \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{\mathsf{throw}(\tilde{s}) \mid P'_{\mathtt{R}}\} \ \mathsf{catch}\ \{Q_{\mathtt{R}}\}\} \ \mathsf{catch}\ \{Q''_{\mathtt{R}}\} \mid \\
\qquad \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{\mathsf{throw}(\tilde{s}) \mid P'_{\mathtt{RS}}\} \ \mathsf{catch}\ \{Q_{\mathtt{RS}}\}; P'\} \ \mathsf{catch}\ \{Q''_{\mathtt{RS}}\}
\end{array}
$$

where $P' = \mathsf{try}(\tilde{s}'')\{P'_{\mathtt{RS}}\} \ \mathsf{catch}\ \{Q''_{\mathtt{RS}}\}$ and $\tilde{s}' = \tilde{s} \cup \tilde{s}''$.

We apply rule $\lfloor$Thr$\rfloor$ obtaining:

$$
\begin{array}{l}
\mathsf{throw}(\tilde{s}) \vdash \mathtt{ResolverTR} \mid \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{P'_{\mathtt{R}}\} \ \mathsf{catch}\ \{Q_{\mathtt{R}}\}\} \ \mathsf{catch}\ \{Q''_{\mathtt{R}}\} \mid \\
\qquad \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s})\{P'_{\mathtt{RS}}\} \ \mathsf{catch}\ \{Q_{\mathtt{RS}}\}; P'\} \ \mathsf{catch}\ \{Q''_{\mathtt{RS}}\}.
\end{array}
$$

Then, we apply rule $\lfloor$RThr$\rfloor$ twice to reduce the inner try-catch blocks:

$$
\begin{array}{l}
\mathsf{throw}(\tilde{s}) \vdash \mathtt{ResolverTR} \mid \ \mathsf{try}(\tilde{s}')\{Q_{\mathtt{R}}\} \ \mathsf{catch}\ \{Q''_{\mathtt{R}}\} \mid \\
\qquad \mathsf{try}(\tilde{s}')\{Q_{\mathtt{RS}}; P'\} \ \mathsf{catch}\ \{Q''_{\mathtt{RS}}\}.
\end{array}
$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

We recall that the handlers have the shape

$$Q_j = \mathsf{try}(\tilde{s})\{\mathsf{if}\ \mathtt{test} = h\ \mathsf{then}\ s_i^1!\langle h\rangle\ \mathsf{else}\ \ s_i^1!\langle 0\rangle; s_j^1?(x).; Q_j'\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}')\}$$

while the handler of the resolver has the shape

$$Q_1 = \mathsf{try}(\tilde{s})\{s_1^1?(x_2)\ldots s_n^1?(x_n).;\ \mathsf{selects}\ h \in H.\ \ s_n^1!\langle h\rangle \ldots s_1^1!\langle h\rangle\}\ \mathsf{catch}\ \{\mathbf{0}\}.$$

Then, we have:

$$\begin{array}{rl}\mathsf{throw}(\tilde{s}) \vdash & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{s_1^1?(x_1); s_2^1?(x_2)\ldots\}\ \mathsf{catch}\ \{\mathbf{0}\}\}\ \mathsf{catch}\ \{\mathbf{0}\}\ \mid \\ & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathsf{if}\ldots\}\ \mathsf{catch}\ \{\tilde{s}^2, \tilde{s}''\}\}\ \mathsf{catch}\ \{Q_{\mathtt{R}}''\}\ \mid \\ & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathsf{if}\ldots\}\ \mathsf{catch}\ \{\tilde{s}^2, \tilde{s}''\}; P'\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}''\}\ \mid \ldots\end{array}$$

we apply both rule [SEND] and rule [REC] twice then we have:

$$\begin{array}{rl}\mathsf{throw}(\tilde{s}) \vdash & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\ \mathsf{algorithm\ determining}\ h \in H. \\ & \quad s_1!\langle h\rangle; \ldots; s_n!\langle h\rangle\}\ \mathsf{catch}\ \{\mathbf{0}\}\}\ \mathsf{catch}\ \{\mathbf{0}\}\ \mid \\ & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{s_i?(x); Q_{\mathtt{R}}'\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}\}\ \mathsf{catch}\ \{Q_{\mathtt{R}}\}\ \mid \\ & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{s_j?(x); Q_{\mathtt{R}}'\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}; P'\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}\}\ \mid \ldots\end{array}$$

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 in Section 6, the handler of the inner try-catch blocks above 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.

1. The execution of the handlers terminates without problems:

$$\begin{array}{rl}\mathsf{throw}(\tilde{s}) \vdash & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathbf{0}\}\}\ \mathsf{catch}\ \{\mathbf{0}\}\ \mid \\ & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}\}\ \mathsf{catch}\ \{Q_{\mathtt{R}}''\}\ \mid \\ & \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}''\}\ \mid \ldots\end{array}$$

and we apply rule [ZTHR] the execution goes on with the next nested action `grab-press-from-plate` in which `RobotSensor`, `Press` and `PressSensor` cooperate:

$$\varnothing \vdash \mathsf{try}(\tilde{s}')\{P'\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}''\}\mathtt{Press} \mid \mathtt{PressSensor} \mid \ldots$$

2. Something goes wrong during the general handlers execution (for instance $Q_{\mathtt{R}}' \longrightarrow^* \mathsf{throw}(\tilde{s}^1)$).

$$\begin{array}{l}\mathsf{throw}(\tilde{s}) \vdash \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathsf{throw}(\tilde{s}^1)\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}\}\ \mathsf{catch}\ \{Q_{\mathtt{R}}''\}\ \mid \\ \qquad\qquad \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{Q_{\mathtt{RS}}\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}; P'\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}''\},\end{array}$$

we apply rule [THR]:

$$\begin{array}{l}\mathsf{throw}(\tilde{s}), \mathsf{throw}(\tilde{s}^1) \vdash \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{\mathbf{0}\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}\}\ \mathsf{catch}\ \{Q_{\mathtt{R}}''\}\ \mid \\ \qquad\qquad \mathsf{try}(\tilde{s}')\{\mathsf{try}(\tilde{s}^1)\{Q_{\mathtt{RS}}\}\ \mathsf{catch}\ \{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}; P'\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}''\},\end{array}$$

we apply rule [RTHR] twice to reduce $\mathsf{throw}(\tilde{s}^1)$ in Σ:

$$\begin{array}{l}\mathsf{throw}(\tilde{s}), \mathsf{throw}(\tilde{s}^1) \vdash \mathsf{try}(\tilde{s}')\{\mathsf{throw}(\tilde{s}^2, \tilde{s}'')\}\ \mathsf{catch}\ \{Q_{\mathtt{R}}''\}\ \mid \\ \mathsf{try}(\tilde{s}')\{\mathsf{throw}(\tilde{s}^2, \tilde{s}''); P'\}\ \mathsf{catch}\ \{Q_{\mathtt{RS}}''\}\end{array}$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

the handler alerts the enclosing action by signalling a throw on channels $\tilde{s}^2, \tilde{s}''$ then rule $\lfloor$Thr$\rfloor$ is applied putting $\mathsf{throw}(\tilde{s}^2, \tilde{s}'')$:

$$\mathsf{throw}(\tilde{s}), \mathsf{throw}(\tilde{s}^1), \mathsf{throw}(\tilde{s}^2, \tilde{s}'') \vdash \mathsf{try}(\tilde{s}')\{\mathbf{0}\} \ \mathsf{catch} \ \{Q''_{\mathtt{R}}\} \mid \mathsf{try}(\tilde{s}')\{P'\} \ \mathsf{catch} \ \{Q''_{\mathtt{RS}}\}.$$

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

$$\mathsf{throw}(\tilde{s}), \mathsf{throw}(\tilde{s}^1), \mathsf{throw}(\tilde{s}^2, \tilde{s}'') \vdash Q_{\mathtt{R}} \mid Q_{\mathtt{RS}} \mid Q_{\mathtt{P}} \mid Q_{\mathtt{PS}}.$$

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

Appendix C. Properties: Technical Aspects and Proofs

C.1. *Proof of subject reduction and communication safety*

In order to prove *subject reduction* and *communication safety,* we need to extend the typing rules to include those for message queues. We adopt the same technique as in Honda *et al.* (2008), therefore we borrow and adapt some useful definitions, which follow:

Definition C.1 (linearity). G is *linear* if, whenever $\mathsf{n}_i = \mathtt{p} \to \mathtt{p}'_{\mathtt{i}} : k$ $(i = 1, 2)$ are in G for some k and do not occur in different branches of a branching, nor one in the default and the other in the handler part of an exception, then both input and output dependencies exist from n_1 to n_2, or, if not, both exist from n_2 to n_1. If G carries other global types, we inductively demand the same.

Local types are considered up to the following isomorphism (closed under all type constructors). We assume $k \neq k'$, $m \in I$ and $n \in J$.

$$k!\langle S\rangle; k'!\langle S'\rangle; T \ \approx \ k'!\langle S'\rangle; k!\langle S\rangle; T \tag{C1}$$

$$k \oplus \{l_i : k' \oplus \{l'_j : T_{ij}\}_{j\in J}\}_{i\in I} \ \approx \ k' \oplus \{l'_j : k \oplus \{l_i : T_{ij}\}_{i\in I}\}_{j\in J} \tag{C2}$$

The equations permute two consecutive outputs with different subjects, capturing asynchrony in communication.

We define *subtyping over end-point types.* We extend the standard session subtyping (Gay and Hole 2005; Honda *et al.* 2008) to deal with try-block types and sequencing.

Definition C.2 (subtyping over end-point types). $\leqslant_{\mathsf{sub}}$ is the maximal fixed point of the function $\mathcal{S}$ that maps each binary relation $\mathcal{R}$ on end-point types as regular trees to $\mathcal{S}(\mathcal{R})$ given as:

— If $A \ \mathcal{R} \ A'$, then $k!\langle S\rangle.A \ \mathcal{S}(\mathcal{R}) \ k!\langle S\rangle.A'$ and $k?\langle S\rangle.A \ \mathcal{S}(\mathcal{R}) \ k?\langle S\rangle.A'$.
— If $A_i \ \mathcal{R} \ A'_i$ for each $i \in I \subset J$, then $\oplus\{l_i : A_i\}_{i\in I} \ \mathcal{S}(\mathcal{R}) \ \oplus\{l_j : A'_j\}_{j\in J}$, and $\&\{l_j : A_j\}_{i\in J} \ \mathcal{S}(\mathcal{R}) \ \&\{l_i : A'_i\}_{i\in I}$.
— If $\alpha \ \mathcal{R} \ \alpha'$ and $\beta \ \mathcal{R} \ \beta'$, then $\{\!|\tilde{k}, \alpha, \beta|\!\} \ \mathcal{S}(\mathcal{R}) \ \{\!|\tilde{k}, \alpha', \beta'|\!\}$.
— If $\alpha \ \mathcal{R} \ \alpha'$, then $\alpha;\beta \ \mathcal{S}(\mathcal{R}) \ \alpha';\beta$.
— If $\beta \ \mathcal{R} \ \beta'$, then $\alpha;\beta \ \mathcal{S}(\mathcal{R}) \ \alpha;\beta'$.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

The typability in the original system in Section 4 and the one in the runtime type system coincide for processes without runtime elements.

Proposition C.3. Let P be a process with no queues and no ν-bound session channels, and Δ be without a type context. Then $\Gamma \vdash P \triangleright \Delta$ iff $\Gamma \vdash P \triangleright_{\varnothing} \Delta$ without using (Subs).

Proof. By definition, P is without queues and without bound channels. We show two implications.

1. $\Gamma \vdash P \triangleright \Delta$ implies $\Gamma \vdash P \triangleright_{\varnothing} \Delta$: suppose P is typable in the static typing system. Since the typing rules for runtime processes subsume the static rules, they can type P with the same derivation.
2. $\Gamma \vdash P \triangleright_{\varnothing} \Delta$ without (Subs) implies $\Gamma \vdash P \triangleright \Delta$: suppose P is typable in the refined system as $\Gamma \vdash P \triangleright_{\varnothing} \Delta$ without type contexts in Δ and without using (Subs). By the lack of (Subs) in the derivation, the derivation precisely follows the structure of P. We inspect the potential differences between the static rules and the runtime rules.

   Use of type contexts in derivation. Suppose the derivation uses a type context. The only place it can be taken off is (Conc). Since, there is no queue in P this means the type context has been empty as the result of weakening by (Inact). Hence, its use can be taken off from the derivations.

   Use of refined constraints on queue channels in judgements. Since, the only rule which decreases the number of mentioned queue channels in the judgement (as in $\triangleright\tilde{s}$) is (CRes), we know each judgement in the derivation has the $\varnothing$ as its mentioned queue channels. Hence, the constraint on queue channels in (Conc) and other rules are never used.

Thus, this derivation for P in the runtime system offers the derivation in the static type system as is. □

Proposition C.4. Below Δ *is coherent* if $\Delta(\tilde{s})$ is coherent for each $\tilde{s} \in dom(\Delta)$.

1. $\Delta_1 \xrightarrow{\lambda} \Delta_1'$ and $\Delta_1 \asymp \Delta_2$ imply $\Delta_1' \asymp \Delta_2$ and $\Delta_1 \circ \Delta_2 \xrightarrow{\lambda} \Delta_1' \circ \Delta_2$.
2. Let Δ be coherent. Then $\Delta \xrightarrow{\lambda} \Delta'$ implies Δ' is coherent.
3. Let Δ be coherent and $\Delta(\tilde{s}) = [\![G]\!]$. Then $\Delta \xrightarrow{s_k^{\varphi}} \Delta'$ (or $\Delta \xrightarrow{\mathsf{throw}(\tilde{s_k^{\varphi}})} \Delta'$) iff $G \xrightarrow{k^{\varphi}} G'$ (or $G \xrightarrow{\mathsf{throw}(k^{\varphi})} G'$) with $\Delta'(\tilde{s}) = [\![G']\!]$.

Proof. As done in Honda *et al.* (2008). See the long version of Honda *et al.* (2008). □

Proof of theorem 5.8

Proof. (1) is by rule induction (cf. Figure 7).
(2) is also by rule induction (cf. Figures 5 and 6). We show only some significant case. If the last applied rule is (Thr):

$$\Sigma \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{\mathcal{E}[\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})]\}\ \mathsf{catch}\ \{Q\} \longrightarrow \Sigma \cup \mathsf{throw}(\tilde{s}^{\tilde{\varphi}}) \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{\mathcal{E}\}\ \mathsf{catch}\ \{Q\},$$

where

$$\tilde{\psi} \geqslant \tilde{\varphi} \quad \textit{implies} \quad \mathsf{throw}(\tilde{s}^{\tilde{\psi}}, \tilde{r}) \notin \Sigma$$

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

then we have $\Gamma \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{\mathcal{E}[\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})]\}\ \mathsf{catch}\ \{Q\} \triangleright_{\varnothing} \Delta$. By typing rule (Try) we get that $\Delta = \{\tilde{s}' : \llbracket \tilde{s}^{\tilde{\varphi}}, \alpha, \beta \rrbracket @\mathtt{p}\}$ and

$$\Gamma \vdash \mathcal{E}[\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})] \triangleright_{\varnothing} \{\tilde{s}' : \alpha@\mathtt{p}\} \qquad \Gamma \vdash Q \triangleright_{\varnothing} \{\tilde{s}' : \beta@\mathtt{p}\}$$
$$\mathsf{ch}(\mathcal{E}[\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})]) \subseteq \tilde{s}'.$$

It is easy to check by mechanical induction on $\mathcal{E}$ that if $\Gamma \vdash \mathcal{E}[\mathsf{throw}(\tilde{s}^{\tilde{\varphi}})] \triangleright_{\varnothing} \{\tilde{s}' : \alpha@\mathtt{p}\}$, then $\Gamma \vdash \mathcal{E} \triangleright_{\varnothing} \{\tilde{s}' : \alpha@\mathtt{p}\}$, having $\Gamma \vdash \mathsf{throw}(\tilde{s}^{\tilde{\varphi}}) \triangleright_{\varnothing} \varnothing$, by typing rule (Throw).

Therefore, by applying rule (Try) we get the result: $\Gamma \vdash \mathsf{try}(\tilde{s}')\{\mathcal{E}\}\ \mathsf{catch}\ \{Q\} \triangleright_{\varnothing} \Delta$.

If the last applied rule is (RThr):

$$\Sigma, \mathsf{throw}(\tilde{s}^{\tilde{\psi}'}) \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{P\}\ \mathsf{catch}\ \{Q\} \mid (\textstyle\prod_i s_i[\varphi_i] : L_i)_{\{s_i^{\varphi_i} \in \mathsf{ch}(P)\}} \longrightarrow$$
$$\Sigma, \mathsf{throw}(\tilde{s}^{\tilde{\psi}'}) \vdash Q \mid (\textstyle\prod_i s_i[\varphi_i] : \varnothing)_{\{s_i^{\varphi_i} \in \mathsf{ch}(P)\}}$$

where

$$\tilde{\psi}' \geqslant \tilde{\varphi} \ \textit{and}$$
$$\left(\tilde{s} = \tilde{z}, \tilde{s}' \ \textit{and}\ \mathsf{throw}(\tilde{z}^{\tilde{\psi}}) \in \Sigma \ \textit{and}\ \tilde{\psi} \geqslant \tilde{\varphi}'\right) \quad \textit{implies} \quad \mathsf{try}(\tilde{z}^{\tilde{\varphi}'}) \ldots \notin P$$

and we have that $\Gamma \vdash \mathsf{try}(\tilde{s}^{\tilde{\varphi}})\{P\}\ \mathsf{catch}\ \{Q\} \triangleright_{\varnothing} \Delta$. By rule (Try) we get that $\Delta = \{\tilde{s}' : \llbracket \tilde{s}^{\tilde{\varphi}}, \alpha, \beta \rrbracket @\mathtt{p}\}$ and $\Gamma \vdash Q \triangleright_{\varnothing} \{\tilde{s}' : \beta@\mathtt{p}\}$.

(3) immediate from (2). □

Proof of Theorem 5.9

Proof.

1. Let $P \equiv (\nu \tilde{n})(P_0 \mid s[\varphi] : L \mid Q)$ where P_0 does not contain a queue and Q only contains queues (by Proposition 5.3). By easy rule induction, we can check that P_0 has either a single active prefix or a pair of a receiving active prefix and an emitting active prefix. So we have three cases:
   - if $P_0\langle s^{\varphi}?\rangle$ and there is no other active prefixes at s, then since there is a redex in P either the sending at s^{φ} has already been performed and the queue cannot be empty, or the sending at s^{φ} has been suppressed because of a thrown exception and $\mathsf{try}(\tilde{r}')\{R\}\ \mathsf{catch}\ \{Q'\}$ is the redex;
   - if $P_0\langle s^{\varphi}!\rangle$ and there is no other active prefixes at s, then this gives us a redex;
   - if $P_0\langle s^{\varphi}!\rangle$ and $P_0\langle s^{\varphi}?\rangle$, then at least the former gives a redex but the latter can also give a redex.
2. For points a.i and b.i, we can proceed in a standard way, noticing that if P satisfies the stated condition then we can write $P \equiv \mathcal{E}'[s^{\varphi} : L \mid R]$ and $S = s[\varphi] : L \mid R$ form a redex, with the same typing by Theorem 5.8 (1). Then the pair of active prefixes at s in the tying of S should be complementary. The rest is by the direct correspondence between the type constructors and the prefixes. Points a.ii and b.ii follow from 1.a.

□

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Proof of corollary 5.10

Proof. In (1), the conclusion $\Gamma \vdash P' \triangleright_{\tilde{t}} \Delta'$ where $\Delta = \Delta'$ or $\Delta \xrightarrow{s_k^{\varphi}} \Delta'$ follows directly from the subject reduction theorem 5.8(2). Then second conclusions $G \xrightarrow{k} G'$ and $[\![G']\!] = \Delta'(\tilde{s})$ follow directly from Proposition C.4(3). If not, a sender puts some value in the queue or an exception may be thrown on k. Hence, (2) obviously holds. □

C.2. **Proof of progress**

Below we safely confuse a channel in a typing and the corresponding free session channel of a process.

Proposition C.5. Let P_0 be simple and $P_0 \to^* P$. Then for each prefix with a shared name in P, say $a[\mathrm{p}](\tilde{s}).P'$ or $\overline{a}[2..n](\tilde{s}).P'$, there is no free session channels in P' except $\tilde{s}$.

Proof. The proof is done in two parts: first we prove that if P is simple then for each prefix with a shared name in P, say $a[\mathrm{p}](\tilde{s}).P'$ or $\overline{a}[2..n](\tilde{s}).P'$, there is no free session channels in P' except $\tilde{s}$. Then, we prove that if P is simple and $P \to P'$ then P' is again simple. The first point is proved by mechanical induction on typing rules. For the second point suppose a derivation of P is simple. By the proof of Theorem 5.8, if $P \to P'$ then we have essentially the same derivation for both P and P' except:

— taking off the lost pair of prefixes from the one of P (three pair of prefix rules);
— one of the branches is chosen (conditional);
— copying some part from the derivation for P to the one of P' (for recursion);
— taking off the last derivation step if $P = \mathsf{try}(\tilde{r})\{R\}\ \mathsf{catch}\ \{P'\}$.

In each case clearly the simplicity of the derivation for P implies the one of P'. □

Lemma C.6. Let $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$ and P is simple. If there is an active receiving (resp. active emitting) prefix in Δ at s^{φ} and none of prefixes at s^{φ} in P is under a prefix at a shared name or under an `if`-branch, then $P\langle s^{\varphi}?\rangle$ (resp. either $P\langle s^{\varphi}!\rangle$ or the queue at s^{φ} is not empty).

Proof. By rule induction. □

Proof of Proposition 5.14. Let P be simple, queue-full and well linked, and $\Gamma \vdash P \triangleright_{\tilde{r}} \Delta$ such that Δ is coherent. Without loss of generality, we can assume P does not have hidings (we can just take off and the result is again simple, queue-full, well linked and coherent). Since Δ is coherent, if Δ contains any prefix then, by Proposition C.5, it should form a redex (together with another prefix to form the image of a merge set) . By Lemma C.6 and Theorem 5.9 (1,2) and by the well linkedness, either there is an if-branch above the prefix or P has an active prefix (or prefixes) at s_k in P or there is a thrown exception on $\tilde{s_k}$ and $s_k \in \tilde{s_k}$. For the former, this if-branch itself cannot be under any prefix since that violates the activeness at s_k in Δ. So this if-branch can reduce. If not then by Lemma C.6 there are the following cases:

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

a. $P \equiv \mathcal{E}[Q\langle s^{\varphi}!\rangle \mid s^{\varphi} : L \mid R\langle s^{\varphi}?\rangle]$, in which case there is at least one redex in P between the emitting prefix and the queue.
b. $P \equiv \mathcal{E}[s^{\varphi} : L \mid R\langle s^{\varphi}?\rangle]$ with L non-empty, in which case there is a redex between the non-empty queue and the receiving redex.
c. $P \equiv \mathcal{E}[Q\langle s^{\varphi}!\rangle \mid s^{\varphi} : L]$, in which case there is a redex as in a.

Notice that even if there is a thrown exception on $\tilde{s_k}$ and $s_k \in \tilde{s_k}$, since $G \xrightarrow{k} G'$ and $\Delta(\tilde{s}) = [\![G]\!]$, this means that a prefix on k is in G and both complementary actions are available in $[\![G]\!]$. □

Proof of theorem 5.15. Immediate from Proposition C.5, Lemma C.6 and Proposition 5.14. □

C.3. **Proof of throw confluence**

Proof of theorem 5.17. To satisfy the premises G must be of the form $\mathcal{G}[(\gamma_1|\{\tilde{k}^{\tilde{\varphi}}, \gamma, \gamma'\};\gamma_2)];\mathsf{end}$, where $\mathcal{G}$ is defined as follows:

$$\mathcal{G} ::= [\,] \quad | \quad \{\tilde{k'}^{\tilde{\psi}}, \mathcal{G}, \gamma\} \quad | \quad \mathcal{G};\gamma \quad | \quad \mathcal{G} \parallel \gamma$$

Without loss of generality, we can assume $\mathcal{G} = [\,]$.

1. If $k \notin \tilde{k}$ then the prefix at k^{φ} cannot be in γ, in γ' nor in γ_2. Hence, k^{φ} must be in γ_1. Then $G \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} (\gamma_1|\gamma';\gamma_2);\mathsf{end} \xrightarrow{k^{\varphi}} (\gamma_1'|\gamma';\gamma_2);\mathsf{end}$, and $G \xrightarrow{k^{\varphi}} (\gamma_1'|\{\tilde{k}^{\tilde{\varphi}}, \gamma, \gamma'\};\gamma_2);\mathsf{end} \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} (\gamma_1'|\gamma';\gamma_2);\mathsf{end}$.
2. If $k \in \tilde{k}$ then $k^{\varphi} \in \gamma$, then $(\gamma_1|\{\tilde{k}^{\tilde{\varphi}}, \gamma, \gamma'\};\gamma_2);\mathsf{end} \xrightarrow{k^{\varphi}} \xrightarrow{\mathsf{throw}(\tilde{k}^{\tilde{\varphi}})} (\gamma_1|\gamma';\gamma_2);\mathsf{end}$.

□

References

Baeten, J. C. M. and Wejland, W. (1990) *Process Algebra*, Cambridge University Press.

Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M. and Yoshida, N. (2008) Global progress in dynamically interleaved multiparty sessions. In: International Conference on Concurrency Theory . *Springer Lecture Notes in Computer Science* **5201** 418–433.

Bocchi, L., Laneve, C. and Zavattaro, G. (2003) A calculus for long-running transactions. In: Proceedinds of Formal Methods for Open Object-Based Distributed Systems. *Springer Lecture Notes in Computer Science* **2884** 124–138.

Boreale, M., Bruni, R., Nicola, R. and Loreti, M. (2008) Sessions and pipelines for structured service programming. In: Barthe, G. and Boer, F. (eds.) Proceedings of Formal Methods for Open Object-Based Distributed Systems, *Lecture Notes in Computer Science*, **5051** 19–38.

Bravetti, M. and Zavattaro, G. (2009) On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, **19** (3) 565–599.

Capecchi, S., Giachino, E. and Yoshida, N. (2010) Global escape in multiparty sessions. In: Proceedings of Foundations of Software Technology and Theoretical Computer Science . *Leibniz International Proceedings in Informatics* **8** Schloss Dagstuhl–Leibniz-Zentrum für Informatik 338–351.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Carbone, M. (2009) Session-based choreography with exceptions. *Electronic Notes in Theoretical Computer Science* **241** 35–55.

Carbone, M., Honda, K. and Yoshida, N. (2008) Structured interactional exceptions for session types. In: 19th International Conference on Concurrency Theory. *Springer Lecture Notes in Computer Science* **5201** 402–417.

Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G. and Ross-Talbot, S. (2006) A theoretical basis of communication-centred concurrent programming. WCD-Working Note.

Castor Filho, F., Romanovsky, A. and Rubira, C. M. F. (2009) Improving reliability of cooperative concurrent systems with exception flow analysis. *Journal Systems and Software* **82** 874–890.

Demangeon, R. and Honda, K. (2012) Nested protocols in session types. In: Koutny, M. and Ulidowski, I. (eds.)Proceedings of International Conference on Concurrency Theory . *Springer Lecture Notes in Computer Science* **7454** 272–286.

Deniélou, P.-M. and Yoshida, N. (2010) Buffered communication analysis in distributed multiparty sessions. In: Proceedings of International Conference on Concurrency Theory . *Springer Lecture Notes in Computer Science* **6269** 343–357.

Gay, S. and Hole, M. (2005) Subtyping for session types in the Pi-calculus. *Acta Informatica* **42** (2/3) 191–225.

Honda, K., Hu, R., Neykova, R., Chen, T.-C., Demangeon, R., Deniélou, P.-M. and Yoshida, N. (2012) Structuring communication with session types. In: Proceedings of Concurrent Objects and Beyond. *Springer Lecture Notes in Computer Science* **8665** 105–127.

Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C. and Yoshida, N. (2011) Scribbling interactions with a formal foundation. In: International Conference on Distributed Computing and Internet Technology. *Springer Lecture Notes in Computer Science* **6536** 55–75.

Honda, K., Vasconcelos, V. T. and Kubo, M. (1998) Language primitives and type disciplines for structured communication-based programming. In: Proceedings of European Symposium on Programming. *Springer Lecture Notes in Computer Science* **1381** 22–138.

Honda, K., Yoshida, N. and Carbone, M. (2008) Multiparty asynchronous session types. In: Principles of Programming Languages. ACM **43** 273–284.

Hu, R., Kouzapas, D., Pernet, O., Yoshida, N. and Honda, K. (2010) Type-safe eventful sessions in Java. In: Proceedings of European Conference on Object-Oriented Programming. *Springer Lecture Notes in Computer Science* **6183** 329–353.

Hu, R., Neykova, R., Yoshida, N. and Demangeon, R. (2013) Practical interruptible conversations: Distributed dynamic verification with session types and Python. In: 4th International Conference on Runtime Verification *Springer Lecture Notes in Computer Science* **8174** 130–148.

Hu, R., Yoshida, N. and Honda, K. (2008) Session-based distributed programming in java. In: Proceedings of European Conference on Object-Oriented Programming. *Springer Lecture Notes in Computer Science* **5142** 516–541.

Jakšić, S. and Padovani, L. (2012) Exception handling for copyless messaging. In: *Proceedings of the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ACM **84** 151–162.

Lanese, I. and Montesi, F. (2010) Error handling: From theory to practice. In: Margaria, T. and Steffen, B. (eds.) *Leveraging Applications of Formal Methods, Verification, and Validation*. volume 6416 Springer Berlin / Heidelberg 66–81.

Lanese, I., Vaz, C. and Ferreira, C. (2010) On the expressive power of primitives for compensation handling. In: Proceedings of European Symposium on Programming 366–386.

Lapadula, A., Pugliese, R. and Tiezzi, F. (2007) A calculus for orchestration of web services. In: Proceedings of European Symposium on Programming. *Springer Lecture Notes in Computer Science* **4421** 33–47.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press

Lewerentz, C. and Lindner, T. (eds.) (1995) *Formal Development of Reactive Systems - Case Study Production Cell*, London, UK. Springer-Verlag.

Neykova, R., Yoshida, N. and Hu, R. (2013). SPY: Local verification of global protocols. In: 4th International Conference on Runtime Verification *Springer Lecture Notes in Computer Science* **8174** 358–363.

Rubira, C. M. F., de Lemos, R., Ferreira, G. R. M. and Filho, F. C. (2005) Exception handling in the development of dependable component-based systems. *Software Practice and Experience* **35** (3) 195–236.

Rubira, C. M. F. and Wu, Z. (1995). Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, IEEE Computer Society 499.

Takeuchi, K., Honda, K. and Kubo, M. (1994). An interaction-based language and its typing system. In: Proceedings of Parallel Architectures and Languages Europe. *Springer Lecture Notes in Computer Science* **817** 398–413.

Tartanoglu, F., Issarny, V., Romanovsky, A. and Levy, N. (2003) Coordinated forward error recovery for composite web services. *IEEE Symposium on Reliable Distributed Systems*, 167.

Vieira, H. T., Caires, L. and Seco, J. C. (2008) The conversation calculus: A model of service-oriented computation. In: Proceedings of European Symposium on Programming. *Springer Lecture Notes in Computer Science* **4960** 269–283.

Xu, J., Romanovsky, A. and Randell, B. (1998) Coordinated exception handling in distributed object systems: From model to system implementation. In: *Proceedings of the The 18th International Conference on Distributed Computing Systems*, Washington, DC, USA, IEEE Computer Society 12–21.

https://doi.org/10.1017/S0960129514000164 Published online by Cambridge University Press