FTMPST: FAULT-TOLERANT MULTIPARTY SESSION TYPES

KIRSTIN PETERS ●[a], UWE NESTMANN ●[b], AND CHRISTOPH WAGNER[c]

[a] Augsburg University, Germany
   e-mail address: kirstin.peters@uni-a.de

[b] TU Berlin, Germany
   e-mail address: uwe.nestmann@tu-berlin.de

[c] Paessler AG, Germany

ABSTRACT. Multiparty session types are designed to abstractly capture the structure of communication protocols and verify behavioural properties. One important such property is progress, i.e., the absence of deadlock. Distributed algorithms often resemble multiparty communication protocols. But proving their properties, in particular termination that is closely related to progress, can be elaborate. Since distributed algorithms are often designed to cope with faults, a first step towards using session types to verify distributed algorithms is to integrate fault-tolerance.

We extend multiparty session types to cope with system failures such as unreliable communication and process crashes. Moreover, we augment the semantics of processes by failure patterns that can be used to represent system requirements (as, e.g., failure detectors). To illustrate our approach we analyse a variant of the well-known rotating coordinator algorithm by Chandra and Toueg.

1. INTRODUCTION

Multi-Party Session Types (MPST) are used to statically ensure correctly coordinated behaviour in systems without global control ([HYC16, CDCPY15]). One important such property is progress, i.e., the absence of deadlock. Like with every other static typing approach, the main advantage is their efficiency, i.e., they avoid the problem of state space explosion. MPST are designed to abstractly capture the structure of communication protocols. They describe global behaviours as *sessions*, i.e., units of conversations [HYC16, BCD+08, BHTY10]. The participants of such sessions are called *roles*. *Global types* specify protocols from a global point of view. These types are used to reason about processes formulated in a *session calculus*.

Distributed algorithms (DA) very much resemble multiparty communication protocols. An essential behavioural property of DA is termination [Tel94, Lyn96], despite failures, but it is often elaborate to prove. It turns out that progress (as provided by MPST) and termination (as required by DA) are closely related.

Many DA were designed in a fault-tolerant way, in order to work in environments, where they have to cope with system failures—be it links dropping messages or processes crashing. Gärtner [Gär99] suggested four different forms of fault-tolerance, depending on whether the safety and liveness requirements are met, or not. An algorithm is called *masking* in the (best) case that both properties hold while tolerating faults transparently, i.e., without further intervention by the programmer. It is called *non-masking*, however, if faults are dealt with explicitly in order to cope with unsafe states, while still guaranteeing liveness. The *fail-safe* case then captures algorithms that remain safe, but not live. (The fourth form is just there for completeness; here neither safety nor liveness is guaranteed.) We focus on masking fault-tolerant algorithms.

While the detection of conceptual design errors is a standard property of type systems, proving correctness of algorithms despite the occurrence of system failures is not. Likewise, traditional MPST do not cover fault tolerance or failure handling. There are several approaches to integrate explicit failure handling in MPST (e.g. [CHY08, CGY16, CVB+16, VCE+18, DHH+15, APN17]). These approaches are sometimes enhanced with recovery mechanisms such as [CDCG17] or even provide algorithms to help find safe states to recover from as in [NY17]. Many of these approaches introduce nested TRY-and-CATCH-blocks and a challenge is to ensure that all participants are consistently informed about concurrent THROWS of exceptions. Therefore, exceptions are propagated within the system. Though explicit failure handling makes sense for high-level applications, the required message overhead is too inefficient for many low-level algorithms. Instead these low-level algorithms are often designed to tolerate a certain amount of failures. Since we focus on the communication structure of systems, additional messages as reaction to faults (e.g. to propagate faults) are considered *non-masking* failure handling. In contrast, we expect masking fault-tolerant algorithms to cope without messages triggered by faults. We study how much unhandled failures a well-typed system can tolerate, while maintaining the typical properties of MPST.

We propose a variant of MPST with unreliable interactions and augment the semantics to also represent failures such as message loss and crashing processes, as well as more abstract concepts of fault-tolerant algorithms such as the possibility to suspect a process to be faulty. To guide the behaviour of unreliable communication, the semantics of processes uses failure patterns that are not defined but *could* be instantiated by an application. This allows us to cover requirements on the system—as, e.g., a bound on the number of faulty processes—as well as more abstract concepts like failure detectors. It is beyond the scope of this paper to discuss *how* failure patterns could be implemented.

1.1. **Related Work.** Type systems are usually designed for failure-free scenarios. An exception is [KGG14] that introduces unreliable broadcast, where a transmission can be received by multiple receivers but not necessarily all available receivers. In the latter case, the receiver is deadlocked. In contrast, we consider fault-tolerant interactions, where in the case of a failure the receiver is *not* deadlocked.

The already mentioned systems in [CHY08, CGY16, CVB+16, VCE+18, DHH+15] extend session types with exceptions thrown by processes within TRY-and-CATCH-blocks, interrupts, or similar syntax. They structurally and semantically encapsulate an unreliable part of a protocol and provide some means to 'detect' a failure and 'react' to it. For example [VCE+18] proposes a variant of MPST with the explicit handling of crash failures. Therefore they coordinate asynchronous messages for run-time crash notifications using a coordinator. Processes in [VCE+18] have access to local failure detectors which eventually detect all

failed peers and do not falsely suspect peers. In contrast we augment the semantics of the session calculus with failure patterns that e.g. allow to implement failure detectors but may also be used to implement system requirements. Exceptions may also describe, why a failure occurred. Here we deliberately do not model causes of failures or how to 'detect' a failure. Different system architectures might provide different mechanisms to do so, for example, by means of time-outs. As is standard for the analysis of DA, our approach allows us to port the verified algorithms on different system architectures that satisfy the necessary system requirements.

Another essential difference is how systems react to faults. In [CGY16], THROW-messages are propagated among nested TRY-and-CATCH-blocks to ensure that all participants are *consistently* informed about concurrent THROWS of exceptions. Fault-tolerant DA, however, have to deal with the problem of inconsistency that some part of a system may consider a process/link as crashed, while at the same time the same process/link is regarded as correct by another part. (This is one of the most challenging problems in the design and verification of *fault-tolerant* DA.) The reason is that *distributed* processes usually cannot reliably observe an error on another system part, unless they are informed by some system "device" (like the "coordinator" of [VCE+18] or the "oracle" of [CGY16]). Therefore, abstractions like unreliable failure detectors are used to model this restricted observability which can, for example, be implemented by time-outs. Failure detectors are often considered to be local (see previous paragraph), but they cannot ensure global consistency. Various degrees of consistency, or [un]reliability, of failure detectors are often defined via constraints that are expressed as global temporal properties [CT96] (see also Section 6). Abstract properties, like the communication predicates in the Heard-Of model [CS09], can also be used to specify minimum requirements on system behaviours at a global level of abstraction in order to be able to guarantee correctness properties.

In previous work, we used the above-mentioned failure detector abstractions in the context of (untyped) process calculi [NF03] to verify properties of several algorithms for Distributed Consensus [NFM03, KN09]. Key for the respective proofs was the intricate reconstruction of global state information from process calculus terms, as we later on formalized in [WN14]. We conjecture that MPST could provide proof support in this context, for example, for the methods that apply to these global states. The work by Francalanza and Hennessy [FH07] also uses a process calculus for the analysis of DA, but employs bisimulation proof techniques. In order to do so, however, the intended properties need to be formulated via some global wrapper code, which provides a level of indirection to proofs. This approach suffers from the absence of clear global state information. In contrast, MPST supply useful global (session type) information from scratch.

1.2. **Summary.** The present paper is an extended version of [PNW22a] that additionally contains the proofs of the presented results as well as some additional explanations (also see the technical report [PNW22b]). In Section 2 we give an impression of the forms of fault-tolerant interactions that we consider. Section 3 introduces the syntax of our version of multiparty session types. The semantics of the session calculus is given in Section 4. In Section 5 we provide the typing rules and show that the standard properties that are usually required for multiparty session types versions are valid in our case. Section 6 provides an example of using fault-tolerant multiparty session types by analysing an implementation of a well-known Consensus algorithm.
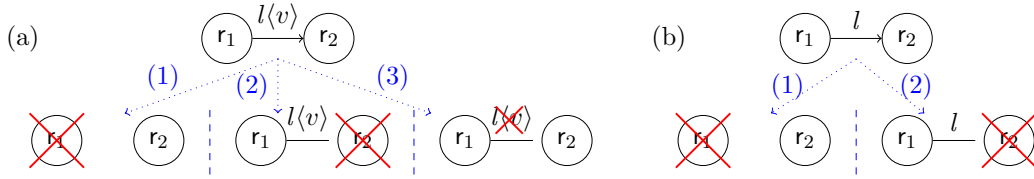
Figure 1: Unreliable Communication (a) and Weakly Reliable Branching (b).

2. Fault-Tolerance in Distributed Algorithms

We consider three sources of failure in an unreliable communication (Figure 1(a)): (1) the sender may crash before it releases the message, (2) the receiver may crash before it can consume the message, or (3) the communication medium may lose the message. The design of a DA may allow it to handle some kinds of failures better than others. Failures are unpredictable events that occur at runtime. Since types consider only static and predictable information, we do not distinguish between different kinds of failure or model their source in types. Instead we allow types, i.e., the specifications of systems, to distinguish between potentially faulty and reliable interactions.

A fault-tolerant algorithm has to solve its task despite such failures. Remember that MPST analyse the communication structure. Accordingly, we need a mechanism to tolerate faults in the communication structure. We want our type system to ensure that a faulty interaction neither blocks the overall protocol nor influences the communication structure of the system after this fault. We consider an unreliable communication as fault-tolerant if a failure does not influence the guarantees for the overall communication structure except for this particular communication. Moreover, if a potentially unreliable communication is executed successfully, then our type system ensures the same guarantees as for reliable communication such as e.g. the absence of communication mismatches.

To ensure that a failure does not block the algorithm, both the receiver and the sender need to be allowed to proceed without their unreliable communication partner. Therefore, the receiver of an unreliable communication is required to specify a default value that, in the case of failure, is used instead of the value the process was supposed to receive. The type system ensures the existence of such default values and checks their sort.

Moreover, we augment unreliable communication with labels that help us to avoid communication mismatches. Assume for instance two subsequent unreliable communications in that values of different sorts, a natural number and a boolean, are transmitted. If the first message with its natural number is lost but the second message containing a Boolean value is transmitted, the receiver could wrongly receive a Boolean value although it still waits for a natural number. To avoid this mismatch, we add a label to unreliable communication, ensure (by the typing rules) that the same label is never associated with different types, and let the semantics inspect the label of a message before reception. Note that this problem, i.e., how to ensure the absence of communication mismatches in the case of unreliable communication, is one of the main challenges in structuring fault-tolerant communication.

Branching in the context of failures is more difficult, because a branch marks a decision point in a specification, i.e., the participants of the session are supposed to behave differently w.r.t. this decision. In an unreliable setting it is difficult to ensure that all participants are informed consistently about such a decision.

Consider a reliable branching that is decided by a process r_1 and transmitted to r_2. If we try to execute such a branching despite failures, we observe that there are again three ways in that this branching can go wrong (Figure 1(b)): (1) The sender may crash before it releases its decision. This will block r_2, because it is missing the information about the branch it should move to. (2) The receiver might crash. (3) The message of r_1 is lost. Then again r_2 is blocked.

Case (2) can be dealt with similar to unreliable communication, i.e., by marking the branching as potentially faulty and by ensuring that a crash of r_2 will not block another process. To deal with Case (1), we declare one of the offered branches as default. If r_1 has crashed, r_2 moves to the default branch. Then r_2 will not necessarily move to the branch that r_1 had in mind before it crashed, but to a valid/specified branch and, since r_1 is crashed, no two processes move to different branches. The main problem is in Case (3). Let r_1 move to a non-default branch and transmit its decision to r_2, this message gets lost, and r_2 moves to the default branch. Now both processes did move to branches that are described by their types; but they are in different branches. This case violates the specification in the type and we want to reject it. More precisely, we consider three levels of failures in interactions:

Strongly Reliable (r): Neither the sender nor the receiver can crash as long as they are involved in this interaction. The message cannot be lost by the communication medium. This form corresponds to reliable communication as it was described in [ACT97] in the context of distributed algorithms. This is the standard, failure-free case.

Weakly Reliable (w): Both the sender and the receiver might crash at every possible point during this interaction. But the communication medium cannot lose the message.

Unreliable (u): Both the sender and the receiver might crash at every possible point during this interaction and the communication medium might lose the message. There are no guarantees that this interaction—or any part of it—takes place. Here, it is difficult to ensure interesting properties in branching.

We use the subscripts or superscripts r, w, or u to indicate actions of the respective kind.

3. Fault-Tolerant Types and Processes

For clarity, we often distinguish names into *values*, i.e., the payload of messages, *shared channels*, or *session channels* according to their usage; there is, however, no need to formally distinguish between different kinds of names.

We assume that the sets $\mathcal{N}$ of names $a, s, x \ldots$; $\mathcal{R}$ of roles $n, r, \ldots$; $\mathcal{L}$ of labels $l, l_d, \ldots$; $\mathcal{V}_T$ of type variables t; and $\mathcal{V}_P$ of process variables X are pairwise distinct. To simplify the reduction semantics of our session calculus, we use natural numbers as roles (compare to [HYC16]). Sorts S range over $\mathbb{B}, \mathbb{N}, \ldots$. The set $\mathcal{E}$ of expressions $e, v, b, \ldots$ is constructed from the standard Boolean operations, natural numbers, names, and (in)equalities.

Global types specify the desired communication structure from a global point of view. In local types this global view is projected to the specification of a single role/participant. We use standard MPST ([HYC08, HYC16]) extended by unreliable communication and weakly reliable branching (highlighted in blue) in Figure 2.

A new session s with n roles is initialised with $\overline{a}[n](s).P$ and $a[r](s).P$ via the shared channel a. We identify sessions with their unique session channel.

The type $r_1 \to_r r_2:\langle S \rangle.G$ specifies a strongly reliable communication from role r_1 to role r_2 to transmit a value of the sort S and then continues with G. A system with this

Global Types	Local Types	Processes
		$P ::= \overline{a}[\mathsf{n}](s).P$
		$\mid \quad a[\mathsf{r}](s).P$
	$T ::= [\mathsf{r}_2]!_\mathrm{r}\langle\mathrm{S}\rangle.T$	$\mid \quad s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{r}\langle e\rangle.P$
$G ::= \mathsf{r}_1 \to_\mathrm{r} \mathsf{r}_2{:}\langle\mathrm{S}\rangle.G$	$\mid \quad [\mathsf{r}_1]?_\mathrm{r}\langle\mathrm{S}\rangle.T$	$\mid \quad s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{r}(x).P$
	$\mid \quad [\mathsf{r}_2]!_\mathrm{u}l\langle\mathrm{S}\rangle.T$	$\mid \quad s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{u}l\langle e\rangle.P$
$\mid \quad \mathsf{r}_1 \to_\mathrm{u} \mathsf{r}_2{:}l\langle\mathrm{S}\rangle.G$	$\mid \quad [\mathsf{r}_1]?_\mathrm{u}l\langle\mathrm{S}\rangle.T$	$\mid \quad s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{u}l\langle v\rangle(x).P$
	$\mid \quad [\mathsf{r}_2]!_\mathrm{r}\{l_i.T_i\}_{i\in\mathrm{I}}$	$\mid \quad s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{r}l.P$
$\mid \quad \mathsf{r}_1 \to_\mathrm{r} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in\mathrm{I}}$	$\mid \quad [\mathsf{r}_1]?_\mathrm{r}\{l_i.T_i\}_{i\in\mathrm{I}}$	$\mid \quad s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{r}\{l_i.P_i\}_{i\in\mathrm{I}}$
	$\mid \quad [\mathsf{R}]!_\mathrm{w}\{l_i.T_i\}_{i\in\mathrm{I}}$	$\mid \quad s[\mathsf{r},\mathsf{R}]!_\mathrm{w}l.P$
$\mid \quad \mathsf{r} \to_\mathrm{w} \mathsf{R}{:}\{l_i.G_i\}_{i\in\mathrm{I},l_\mathrm{d}}$	$\mid \quad [\mathsf{r}]?_\mathrm{w}\{l_i.T_i\}_{i\in\mathrm{I},l_\mathrm{d}}$	$\mid \quad s[\mathsf{r}_j,\mathsf{r}]?_\mathrm{w}\{l_i.P_i\}_{i\in\mathrm{I},l_\mathrm{d}}$
$\mid \quad G_1 \parallel G_2$		$\mid \quad P_1 \mid P_2$
$\mid \quad (\mu t)G \mid t \mid \mathtt{end}$	$\mid \quad (\mu t)T \mid t \mid \mathtt{end}$	$\mid \quad (\mu X)P \mid X \mid \mathbf{0}$
		$\mid \quad \mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$
		$\mid \quad (\nu x)P \mid \perp$
	$\mid \quad [\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T\rangle.T'$	$\mid \quad s[\mathsf{r}_1,\mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P$
$\mid \quad \mathsf{r}_1 \to \mathsf{r}_2{:}\langle s'[\mathsf{r}]{:}T\rangle.G$	$\mid \quad [\mathsf{r}_1]?\langle s'[\mathsf{r}]{:}T\rangle.T'$	$\mid \quad s[\mathsf{r}_2,\mathsf{r}_1]?(\!(s'[\mathsf{r}])\!).P$
		$\mid \quad s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}$
Message Types		Messages
$\mathsf{mt} ::= \langle\mathrm{S}\rangle^\mathrm{r} \mid l\langle\mathrm{S}\rangle^\mathrm{u} \mid l^\mathrm{r} \mid l^\mathrm{w} \mid s[\mathsf{r}]$		$\mathsf{m} ::= \langle v\rangle^\mathrm{r} \mid l\langle v\rangle^\mathrm{u} \mid l^\mathrm{r}$ $\mid l^\mathrm{w} \mid s[\mathsf{r}]$

Figure 2: Syntax of Fault-Tolerant MPST

type will be guaranteed to perform a corresponding action. In a session s this communication is implemented by the sender $s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{r}\langle e\rangle.P_1$ (specified as $[\mathsf{r}_2]!_\mathrm{r}\langle\mathrm{S}\rangle.T_1$) and the receiver $s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{r}(x).P_2$ (specified as $[\mathsf{r}_1]?_\mathrm{r}\langle\mathrm{S}\rangle.T_2$). As result, the receiver instantiates x in its continuation P_2 with the received value.

The type $\mathsf{r}_1 \to_\mathrm{u} \mathsf{r}_2{:}l\langle\mathrm{S}\rangle.G$ specifies an unreliable communication from r_1 to r_2 transmitting (if successful) a label l and a value of type S and then continues (regardless of the success of this communication) with G. The unreliable counterparts of senders and receivers are $s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{u}l\langle e\rangle.P_1$ (specified as $[\mathsf{r}_2]!_\mathrm{u}l\langle\mathrm{S}\rangle.T_1$) and $s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{u}l\langle v\rangle(x).P_2$ (specified as $[\mathsf{r}_1]?_\mathrm{u}l\langle\mathrm{S}\rangle.T_2$). The receiver $s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{u}l\langle v\rangle(x).P_2$ declares a default value v that is used instead of a received value to instantiate x after a failure. Moreover, a label is communicated that helps us to ensure that a faulty unreliable communication has no influence on later actions.

The strongly reliable branching $\mathsf{r}_1 \to_\mathrm{r} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in\mathrm{I}}$ allows r_1 to pick one of the branches offered by r_2. We identify the branches with their respective label. Selection of a branch is by $s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{r}l.P$ (specified as $[\mathsf{r}_2]!_\mathrm{r}\{l_i.T_i\}_{i\in\mathrm{I}}$). Upon receiving l_j, $s[\mathsf{r}_2,\mathsf{r}_1]?_\mathrm{r}\{l_i.P_i\}_{i\in\mathrm{I}}$ (specified as $[\mathsf{r}_1]?_\mathrm{r}\{l_i.T_i\}_{i\in\mathrm{I}}$) continues with P_j.

As discussed in the end of Section 1, the counterpart of branching is weakly reliable and not unreliable. It is implemented by $\mathsf{r} \to_\mathrm{w} \mathsf{R}{:}\{l_i.G_i\}_{i\in\mathrm{I},l_\mathrm{d}}$, where $\mathsf{R} \subseteq \mathcal{R}$ and l_d with $\mathrm{d} \in \mathrm{I}$ is the default branch. We use a broadcast from r to all roles in R to ensure that the sender can influence several participants consistently. Splitting this action to inform the roles in R separately does not work, because we cannot ensure consistency if the sender crashes while performing these subsequent actions. The type system will ensure that no message is

lost. Because of that, all processes that are not crashed will move to the same branch. We often abbreviate branching w.r.t. to a small set of branches by omitting the set brackets and instead separating the branches by $\oplus$, where the last branch is always the default branch. In contrast to the strongly reliable cases, $s[\mathsf{r}, \mathsf{R}]!_\mathrm{w} l.P$ (specified as $[\mathsf{R}]!_\mathrm{w}\{l_i.T_i\}_{i\in\mathrm{I}}$) allows to broadcast its decision to R and $s[\mathsf{r}_j, \mathsf{r}]?_\mathrm{w}\{l_i.P_i\}_{i\in\mathrm{I},l_\mathrm{d}}$ (specified as $[\mathsf{r}]?_\mathrm{w}\{l_i.T_i\}_{i\in\mathrm{I},l_\mathrm{d}}$) defines a default label l_d.

The $\perp$ denotes a process that crashed. Similar to [HYC16], we use message queues to implement asynchrony in sessions. Therefore, session initialisation introduces a directed and initially empty message queue $s_{\mathsf{r}_1\to\mathsf{r}_2}:[\,]$ for each pair of roles $\mathsf{r}_1 \neq \mathsf{r}_2$ of the session s. The separate message queues ensure that messages with different sources or destinations are not ordered, but each message queue is FIFO. Since the different forms of interaction might be implemented differently (e.g. by TCP or UDP), it make sense to further split the message queues into three message queues for each pair $\mathsf{r}_1 \neq \mathsf{r}_2$ such that different kinds of messages do not need to be ordered. To simplify the presentation of examples in this paper and not to blow up the number of message queues, we stick to a single message queue for each pair $\mathsf{r}_1 \neq \mathsf{r}_2$, but the correctness of our type system does not depend on this decision. We have five kinds of messages m and corresponding message types mt in Figure 2—one for each kind of interaction. In strongly reliable communication a value v (of sort S) is transmitted in a message $\langle v\rangle^\mathrm{r}$ of type $\langle\mathsf{S}\rangle^\mathrm{r}$. In unreliable communication the message $l\langle v\rangle^\mathrm{u}$ (of type $l\langle\mathsf{S}\rangle^\mathrm{u}$) additionally carries a label l. For branching only the picked label l is transmitted and we add the kind of branching as superscript, i.e., message/type l^r is for strongly reliable branching and message/type l^w for weakly reliable branching. Finally, message/type $s[\mathsf{r}]$ is for session delegation. A message queue M is a list of messages m and MT is a list of message types mt.

The remaining operators for independence $G \parallel G'$; parallel composition $P \mid P'$; recursion $(\mu t)G$, $(\mu X)P$; inaction $\mathbf{end}$, $\mathbf{0}$; conditionals $\mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$; session delegation $\mathsf{r}_1 \to \mathsf{r}_2{:}\langle s'[\mathsf{r}]{:}T\rangle.G$, $s[\mathsf{r}_1, \mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P$, $s[\mathsf{r}_2, \mathsf{r}_1]?(\!(s'[\mathsf{r}])\!).P$; and restriction $(\nu x)P$ are all standard. As usual, we assume that recursion variables are guarded and do not occur free in types or processes.

In types $(\mu t)G$ and $(\mu t)T$ the type variable t is *bound*. In processes $(\mu X)P$ the process variable X is bound. Similarly, all names in round brackets are bound in the remainder of the respective process, e.g. s is bound in P by $\overline{a}[\mathsf{n}](s).P$ and x is bound in P by $s[\mathsf{r}_1, \mathsf{r}_2]?_\mathrm{r}(x).P$. A variable or name is *free* if it is not bound. Let $\mathrm{FN}(P)$ return the free names of P.

Let *subterm* denote a (type or process) expression that syntactically occurs within another (type or process) term. We use '.' (as e.g. in $\overline{a}[\mathsf{r}](s).P$) to denote sequential composition. In all operators the *prefix* before '.' guards the *continuation* after the '.'. Let $\prod_{1\leq i\leq n} P_i$ abbreviate $P_1 \mid \ldots \mid P_n$.

Let $\mathrm{R}(G)$ return all roles that occur in G. We write $\mathrm{nsr}(G)$, $\mathrm{nsr}(T)$, and $\mathrm{nsr}(P)$, if none of the prefixes in G, T, and P is strongly reliable or for delegation and if P does not contain message queues.

Definition 3.1 (Well-Formedness, Global Type). A global type is *well-formed* if
(1) it neither contains free nor unguarded type variables,
(2) $\mathrm{R}(G) = \{1, \ldots, |\mathrm{R}(G)|\}$,
(3) for all its subterms of the form $\mathsf{r}_1 \to_\mathrm{r} \mathsf{r}_2{:}\langle\mathsf{S}\rangle.G$ or $\mathsf{r}_1 \to_\mathrm{u} \mathsf{r}_2{:}l\langle\mathsf{S}\rangle.G$, we have $\mathsf{r}_1 \neq \mathsf{r}_2$,
(4) for all its subterms of the form $\mathsf{r}_1 \to_\mathrm{r} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in\mathrm{I}}$ or $\mathsf{r} \to_\mathrm{w} \mathsf{R}{:}\{l_i.G_i\}_{i\in\mathrm{I},l_\mathrm{d}}$, we have $\mathsf{r}_1 \neq \mathsf{r}_2$, $\mathsf{r} \notin \mathsf{R}$, $\mathrm{d} \in \mathrm{I}$, and the labels l_i are pairwise distinct, and
(5) for all its subterms of the form $G_1 \parallel G_2$, we have $\mathrm{R}(G_1) \cap \mathrm{R}(G_2) = \emptyset$.

We restrict our attention to well-formed global types.

Definition 3.2 (Well-Formedness, Global Type)**.** A local type is *well-formed* if
(1) it neither contains free nor unguarded type variables and
(2) for all its subterms of the form $[r]!_r\{l_i.T_i\}_{i\in I}$, $[r]?_r\{l_i.T_i\}_{i\in I}$, $[R]!_w\{l_i.T_i\}_{i\in I,l_d}$, or of the form $[R]?_w\{l_i.T_i\}_{i\in I,l_d}$, we have $d \in I$ and the labels l_i are pairwise distinct.

We restrict our attention to well-formed local types.

A session channel and a role together uniquely identify a participant of a session, called an *actor*. A process has an actor $s[r]$ if it has an action prefix on s that mentions r as its first role. Let $A(P)$ be the set of actors of P.

3.1. Examples.
Consider the specification $G_{\text{dice,r}}$ of a simple dice game in a bar

$$(\mu t)3 \rightarrow_r 1:\langle\mathbb{N}\rangle.3 \rightarrow_r 2:\langle\mathbb{N}\rangle.3 \rightarrow_r 1:\{roll.3 \rightarrow_r 2:roll.t, exit.3 \rightarrow_r 2:exit.\mathsf{end}\} \qquad (1)$$

where the dealer Role 3 continues to *roll* a dice and tell its value to player 1 and then to *roll* another time for player 2 until the dealer decides to *exit* the game.

We can combine strongly reliable communication/branching and unreliable communication, e.g. by ordering a drink before each round in $G_{\text{dice,r}}$.

$$(\mu t)3 \rightarrow_u 4:drink\langle\mathbb{N}\rangle.3 \rightarrow_r 1:\langle\mathbb{N}\rangle.3 \rightarrow_r 2:\langle\mathbb{N}\rangle.$$
$$3 \rightarrow_r 1:\{roll.3 \rightarrow_r 2:roll.t, \quad exit.3 \rightarrow_r 2:exit.\mathsf{end}\}$$

where role 4 represents the bar tender and the noise of the bar may swallow these orders. Moreover, we can remove the branching and specify a variant of the dice game in that 3 keeps on rolling the dice forever, but, e.g. due to a bar fight, one of our three players might get knocked out at some point or the noise of this fight might swallow the announcements of role 3:

$$G_{\text{dice,u}} = (\mu t)3 \rightarrow_u 1:roll\langle\mathbb{N}\rangle.3 \rightarrow_u 2:roll\langle\mathbb{N}\rangle.t \qquad (2)$$

To restore the branching despite the bar fight that causes failures, we need the weakly reliable branching mechanism.

$$G_{\text{dice}} = (\mu t)3 \rightarrow_w \{1,2\}: \; play.3 \rightarrow_u 1:roll\langle\mathbb{N}\rangle.3 \rightarrow_u 2:roll\langle\mathbb{N}\rangle.t, \qquad (3)$$
$$\oplus \; end.3 \rightarrow_u 1:win\langle\mathbb{B}\rangle.3 \rightarrow_u 2:win\langle\mathbb{B}\rangle.\mathsf{end}$$

If 3 is knocked out by the fight, i.e., crashes, the game cannot continue. Then 1 and 2 move to the default branch *end*, have to skip the respective unreliable communications, and terminate. But the game can continue as long as 3 and at least one of the players $1,2$ participate.

An implementation of G_{dice} is $P_{\text{dice}} = P_3 \mid P_1 \mid P_2$, where for $i \in \{1,2\}$:

$$P_3 = \overline{a}[3](s).(\mu X)\mathtt{if}\ x_1 \leq 21 \wedge x_2 \leq 21$$

$$\mathtt{then}\ s[3,\{1,2\}]!_w play.s[3,1]!_u roll\langle\mathsf{roll}(x_1)\rangle.s[3,2]!_u roll\langle\mathsf{roll}(x_2)\rangle.X$$
$$\mathtt{else}\ s[3,\{1,2\}]!_w end.s[3,1]!_u win\langle x_1 \leq 21\rangle.s[3,2]!_u win\langle x_2 \leq 21\rangle.\mathbf{0}$$
$$P_i = a[i](s).(\mu X)s[i,3]?_w play.s[i,3]?_u roll\langle x\rangle(x).X \oplus end.s[i,3]?_u win\langle\mathtt{f}\rangle(w).\mathbf{0}$$

Role 3 stores the sums of former dice rolls for the two players in its local variables x_1 and x_2, and $\mathsf{roll}(x_i)$ rolls a dice and adds its value to the respective x_i. Role 3 keeps rolling dice until the sum x_i for one of the players exceeds 21. If both sums x_1 and x_2 exceed 21 in the same round, then 3 wins, i.e., both players receive $\mathtt{f}$; else, the player that stayed below 21 wins

and receives t. The players 1 and 2 use their respective last known sum that is stored in x as default value for the unreliable communication in the branch *play* and f as default value in the branch *end*. The last branch, i.e., *end*, is the default branch.

3.2. **Projection.** Our type system verifies processes, i.e., implementations, against a specification that is a global type. Since processes implement local views, local types are used as a mediator between the global specification and the respective local end points. To ensure that the local types correspond to the global type, they are derived by *projection*. Instead of the projection function described in [HYC16] we use a more relaxed variant of projection as introduced in [YDBH10, CDGH20, vGHH21].

Projection maps global types onto the respective local type for a given role p. The projections of the new global types are obtained straightforwardly from the projection of their respective strongly reliable counterparts:

$$(r_1 \to_\diamond r_2{:}\mathfrak{S}.G)\!\restriction_{\mathsf{p}} \triangleq \begin{cases} [r_2]!_\diamond\mathfrak{S}.G\!\restriction_{\mathsf{p}} & \text{if } \mathsf{p} = r_1 \\ [r_1]?_\diamond\mathfrak{S}.G\!\restriction_{\mathsf{p}} & \text{if } \mathsf{p} = r_2 \\ G\!\restriction_{\mathsf{p}} & \text{otherwise} \end{cases}$$

where either $\diamond = \mathrm{r}$, $\mathfrak{S} = \langle\mathrm{S}\rangle$ or $\diamond = \mathrm{u}$, $\mathfrak{S} = l\langle\mathrm{S}\rangle$ and

$$\big(r_1 \to_\diamond \mathfrak{R}{:}\{l_i.G_i\}_{i\in\mathrm{I}\mathfrak{D}}\big)\!\restriction_{\mathsf{p}} \triangleq \begin{cases} [\mathfrak{R}]!_\diamond\{l_i.G_i\!\restriction_{\mathsf{p}}\}_{i\in\mathrm{I}} & \text{if } \mathsf{p} = r_1 \\ [r_1]?_\diamond\{l_i.G_i\!\restriction_{\mathsf{p}}\}_{i\in\mathrm{I}\mathfrak{D}} & \text{if } \mathfrak{B} \\ \bigsqcup_{i\in\mathrm{I}}(G_i\!\restriction_{\mathsf{p}}) & \text{otherwise} \end{cases}$$

where either $\diamond = \mathrm{r}$, $\mathfrak{R} = r_2$, $\mathfrak{B}$ is $\mathsf{p} = r_2$, $\mathfrak{D}$ is empty or $\diamond = \mathrm{w}$, $\mathfrak{R} = \mathsf{R}$, $\mathfrak{B}$ is $\mathsf{p} \in \mathsf{R}$, $\mathfrak{D}$ is , l_d. In the last case of strongly reliable or weakly reliable branching—when projecting onto a role that does not participate in this branching—we map to $\bigsqcup_{i\in\{1,...,n\}}(G_i\!\restriction_{\mathsf{p}}) = (G_1\!\restriction_{\mathsf{p}}) \sqcup \ldots \sqcup (G_n\!\restriction_{\mathsf{p}})$. The $\sqcup$ allows to unify the projections $G_i\!\restriction_{\mathsf{p}}$ if all of them return the same kind of branching input $[\mathsf{p}]?_\diamond\ldots$ were the respective sets of branches my differ as long as the same label is always followed by the same local type. The operation $\sqcup$ is (similar to [YDBH10]) inductively defined as:

$$T \sqcup T = T$$
$$([r]?_\mathrm{r}\mathrm{I}_1) \sqcup ([r]?_\mathrm{r}\mathrm{I}_2) = [r]?_\mathrm{r}(\mathrm{I}_1 \sqcup \mathrm{I}_2)$$
$$([r]?_\mathrm{w}\mathrm{I}_1) \sqcup ([r]?_\mathrm{w}\mathrm{I}_2) = [r]?_\mathrm{w}(\mathrm{I}_1 \sqcup \mathrm{I}_2) \quad \text{if } \mathrm{I}_1 \text{ and } \mathrm{I}_2 \text{ have the same default branch}$$
$$\mathrm{I} \sqcup \emptyset = \mathrm{I}$$
$$\mathrm{I} \sqcup (\{l.T\} \cup \mathrm{J}) = \begin{cases} \{l.(T' \sqcup T)\} \cup ((\mathrm{I} \setminus \{l.T'\}) \sqcup \mathrm{J}) & \text{if } l.T' \in \mathrm{I} \\ \{l.T\} \cup (\mathrm{I} \sqcup \mathrm{J}) & \text{if } l \notin \mathrm{I} \end{cases}$$

where $T, T' \in \mathcal{T}$ are local types, $\mathrm{I}, \mathrm{I}_1, \mathrm{I}_2, \mathrm{J}$ are sets of branches of local types of the form $l.T$, $l \notin \mathrm{I}$ is short hand for $\nexists T'. \; l.T' \in \mathrm{I}$, and is undefined in all other cases. By the first line, identical types can be merged. By the second and third line, local types for the reception of a branching request can be merged if they have the same prefix and the respective sets of branches can be merged. The third line, for the weakly reliable case, additionally requires that the two sets of branches have the same default branch. The sets of branches, that need to be merged according to the second and third line, contain elements of the form $l.T$, where l is a label and T a local type. The last two lines above inductively define how to merge such sets, i.e., here we overload the operator $\sqcup$ on local types to an operator on sets of branches

of local types. The case distinction in the last line ensures that elements $l.T$ with a label that occurs in only one of the two sets can be kept, but if both sets contain an element with the same label then the respective local types have to be merged for the resulting set.

The mergeability relation $\sqcup$ states that two types are identical up to their branching types, where only branches with distinct labels are allowed to be different. This ensures that if the sender r_1 in $r_1 \rightarrow_r r_2:\{l_i.G_i\}_{i\in I}$ decides to branch then only processes that are informed about this decision can adapt their behaviour accordingly; else projection is **not** defined.

The remaining global types are projected as follows:

$$(r_1 \rightarrow r_2:\langle s[r]:T\rangle.G)\!\upharpoonright_{\mathsf{p}} \triangleq \begin{cases} [r_2]!\langle s[r]:T\rangle.G\!\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} = r_1 \\ [r_1]?\langle s[r]:T\rangle.G\!\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} = r_2 \\ G\!\upharpoonright_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$(G_1 \parallel G_2)\!\upharpoonright_{\mathsf{p}} \triangleq \begin{cases} G_1\!\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} \notin \mathrm{R}(G_2) \\ G_2\!\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p} \notin \mathrm{R}(G_1) \end{cases}$$

$$((\mu t)G)\!\upharpoonright_{\mathsf{p}} \triangleq \begin{cases} G\!\upharpoonright_{\mathsf{p}} & \text{if } t \text{ does not occur in } G \\ (\mu t)G\!\upharpoonright_{\mathsf{p}} & \text{else if } \mathsf{p} \in \mathrm{R}(G) \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad t\!\upharpoonright_{\mathsf{p}} \triangleq t \quad \mathsf{end}\!\upharpoonright_{\mathsf{p}} \triangleq \mathsf{end}$$

The projection of delegation is similar to communication. The projection of $G_1 \parallel G_2$ on p is **not** defined if p occurs on both sides of this parallel composition; it is $G_i\!\upharpoonright_{\mathsf{p}}$ if p occurs in exactly one side $i \in \{1, 2\}$; or it is $(G_1 \parallel G_2)\!\upharpoonright_{\mathsf{p}} = G_1\!\upharpoonright_{\mathsf{p}} = G_2\!\upharpoonright_{\mathsf{p}} = \mathsf{end}$ if p does not occur at all. Recursive types without their recursion variable are mapped to the projection of their recursion body (similar to [CDGH20]), else if p occurs in the recursion body we map to a recursive local type, or else to successful termination. Type variables and successful termination are mapped onto themselves. We denote a global type G as *projectable* if for all $r \in \mathrm{R}(G)$ the projection $G\!\upharpoonright_r$ is defined. We restrict our attention to projectable global types. Projection maps well-formed global types onto the respective local type for a given role p, where the results of projection—if defined—are again well-formed.

Projecting the global type $G_{\mathrm{dice,r}}$ in (1) results in the local types

$$T_{3:\mathrm{dice,r}} = (\mu t)[1]!_r\langle\mathbb{N}\rangle.[2]!_r\langle\mathbb{N}\rangle.[1]!_r\{roll.[2]!_r roll.t, \quad exit.[2]!_r exit.\mathsf{end}\}$$
$$T_{\mathrm{i:dice,r}} = (\mu t)[3]?_r\langle\mathbb{N}\rangle.[3]?_r\{roll.t, \quad exit.\mathsf{end}\}$$

where the types of the two players $T_{1:\mathrm{dice,r}} = T_{2:\mathrm{dice,r}} = T_{\mathrm{i:dice,r}}$ are identical. The projection of the outer branching in $G_{\mathrm{dice,r}}$ on 2 results in $[3]?_r roll.t$ for the first branch and $[3]?_r exit.\mathsf{end}$ for the second branch. These two $[3]?_r$ types are unified by $\sqcup$ into a single $[3]?_r$ type with two branches.

Projection maps G_{dice} in (3) to:

$$T_{3:\mathrm{dice}} = (\mu t)[\{1,2\}]!_w \ play.[1]!_u roll\langle\mathbb{N}\rangle.[2]!_u roll\langle\mathbb{N}\rangle.t$$
$$\oplus \ end.[1]!_u win\langle\mathbb{B}\rangle.[2]!_u win\langle\mathbb{B}\rangle.\mathsf{end}$$
$$T_{\mathrm{i:dice}} = (\mu t)[3]?_w(play.[3]?_u roll\langle\mathbb{N}\rangle.t \oplus end.[3]?_u win\langle\mathbb{B}\rangle.\mathsf{end})$$

where $\mathsf{i} \in \{1,2\}$ and both $T_{\mathrm{i:dice}}$ are obtained by the second case of projection. The type system will ensure that either 3 transmits the request to branch to both players $1, 2$ simultaneously and, since these messages cannot be lost, all players that are not crashed move to same branch or 3 crashes and all remaining players move to the default branch.

Assume instead that 3 can only inform one of the players $1, 2$ at once.

$$(\mu t)3 \to_w \{1\}:\ play.3 \to_u 1{:}roll\langle\mathbb{N}\rangle.3 \to_u 2{:}roll\langle\mathbb{N}\rangle.t$$
$$\oplus\ end.3 \to_u 1{:}win\langle\mathbb{B}\rangle.3 \to_u 2{:}win\langle\mathbb{B}\rangle.\texttt{end}$$

is not projectable, because $\sqcup$ does not allow to unify the projections $[3]?_u roll\langle\mathbb{N}\rangle.t$ and $[3]?_u win\langle\mathbb{B}\rangle.\texttt{end}$ of the two branches of 2. Replacing them by strongly reliable communications implies that neither 3 nor 2 fail. The type

$$(\mu t)3 \to_w \{1\}:\ play.3 \to_u 1{:}roll\langle\mathbb{N}\rangle.3 \to_w \{2\}{:}(play.3 \to_u 2{:}roll\langle\mathbb{N}\rangle.t \oplus end.\texttt{end})$$
$$\oplus\ end.3 \to_u 1{:}win\langle\mathbb{B}\rangle.3 \to_w \{2\}{:}(play.\texttt{end} \oplus end.3 \to_u 2{:}win\langle\mathbb{B}\rangle.\texttt{end})$$

where 3 informs its two players subsequently about the chosen branch is projectable. But it introduces the two additional branches $end.\texttt{end}$ and $play.\texttt{end}$, i.e., 3 is allowed to choose the branches for the players $1, 2$ separately and differently, whereas in (1) as well as in (3) the players $1, 2$ are always in the same branch. Because of that, we allow for broadcast in weakly reliable branching such that 3 can inform both players consistently without introducing additional and not-intended branches.

3.3. **Labels.** We use labels for two purposes: they allow us to distinguish between different branches, as usual in MPST-frameworks, and we assume that they may carry additional runtime information such as timestamps. We do that because we think of labels not only as identifiers for branching, but also as some kind of meta data of messages as they can be often found in communication media or as they are assumed by many distributed algorithms. A prominent example is the use of timestamps in message headers, that allow a receiver to identify outdated messages and to discard them. Thereby, these additional runtime information placed in the label by the sender help the receiver to implement one of the already mentioned failure patterns; namely the one that allows a receiver to skip a message and continue with a default value instead. We will introduce failure patterns with the semantics in the next section. Although it is beyond the scope of this paper to discuss the implementation of failure patterns, we have to provide the technical means to do so.

Allowing for runtime information in labels requires a subtle difference in the way labels are used. A timestamp may be added by the sender to capture the transmission time, but for the receiver it is hard to have this information already present in its label before or during reception. Similarly, types in our static type system should not depend on any runtime information. Hence, in contrast to standard MPST, we do not expect the labels of senders and receivers as well as the labels of processes and types to match exactly. Instead we assume a predicate $\doteq$ that compares two labels and is satisfied if the parts of the labels that do not refer to runtime information correspond. If labels do not contain runtime information, $\doteq$ can be instantiated with equality. We require that $\doteq$ is unambiguous on labels used in types, i.e., given two labels of processes l_P, l'_P and two labels of types l_T, l'_T then $l_P \doteq l'_P \wedge l_P \doteq l_T \Rightarrow l'_P \doteq l_T$ and $l_P \doteq l_T \wedge l_T \neq l'_T \Rightarrow l_P \neq l'_T$.

Of course, the presented type system remains valid if we use labels without additional runtime information. Indeed all presented examples carry in their labels statically available information only. Interestingly, also the static information in labels, that have to coincide for senders and receivers and their types, can be exploited to guide communication. In contrast to standard MPST and to support unreliable communication, our MPST variant will ensure that all occurrences of the same label are associated with the same sort. This helps us in the case of failures to ensure the absence of communication mismatches, i.e., the

type of a transmitted value has to be the type that the receiver expects. The global type $G_{\mathbb{NB}} = 1 \rightarrow_\mathrm{u} 2{:}l_1\langle\mathbb{N}\rangle.1 \rightarrow_\mathrm{u} 2{:}l_2\langle\mathbb{B}\rangle.\texttt{end}$ specifies two subsequent unreliable communications in that values of different sorts are transmitted as discussed in Section 2. If the first message with its natural number is lost but the second message containing a Boolean value is transmitted, the receiver 2 should not wrongly receive a Boolean value although it still waits for a natural number. To avoid this mismatch, we add a label to unreliable communication and ensure (by the typing rules) that the same label is never associated with different types. In the case of $G_{\mathbb{NB}}$, the type system associates l_1 with sort $\mathbb{N}$ and l_2 with sort $\mathbb{B}$ and ensures that $l_1 \neq l_2$. Sine $l_1 \neq l_2$, the reduction rules do not allow the receiver to use the value received in a message with label l_2 for its first communication action, i.e., forces the receiver to first skip its first communication and use a default value before its is allowed to receive the message with label l_2. Here we interpret labels again as some kind of meta data of messages that allow a receiver to use static information in labels to guide reception. In particular, our unreliable communication mechanism exploits such meta data to guarantee strong properties about the communication structure including the described absence of communication mismatches. Since the labels of the sender and the receiver are associated with a unique sort, the type system can then ensure that received values have the expected sort. Similarly, labels are used in [CV10] to avoid communication errors.

4. A Semantics with Failure Patterns

Before we describe the semantics, we introduce substitution and structural congruence as auxiliary concepts. The application of a substitution $\{y/x\}$ on a term A, denoted as $A\{y/x\}$, is defined as the result of replacing all free occurrences of x in A by y, possibly applying alpha-conversion to avoid capture or name clashes. For all names $n \in \mathcal{N} \setminus \{x\}$ the substitution behaves as the identity mapping. We use substitution on types as well as processes and naturally extend substitution to the substitution of variables by terms (to unfold recursions) and names by expressions (to instantiate a bound name with a received value). We assume an evaluation function $\mathrm{eval}(\cdot)$ that evaluates expressions to values.

We use structural congruence to abstract from syntactically different processes with the same meaning, where $\equiv$ is the least congruence that satisfies alpha conversion and the rules:

$$P \mid \mathbf{0} \equiv P \qquad P_1 \mid P_2 \equiv P_2 \mid P_1 \qquad P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3 \qquad (\mu X)\mathbf{0} \equiv \mathbf{0}$$
$$(\nu x)\mathbf{0} \equiv \mathbf{0} \qquad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \qquad (\nu x)\,(P_1 \mid P_2) \equiv P_1 \mid (\nu x)P_2 \quad \text{if } x \notin \mathrm{FN}(P_1)$$

The reduction semantics of the session calculus is defined in the Figures 3 and 4, where we follow [HYC16]: session initialisation is synchronous and communication within a session is asynchronous using message queues. The rules are standard except for the five failure pattern and two rules for system failures: (Crash) for *crash failures* and (ML) for *message loss*. *Failure patterns* are predicates that we deliberately choose not to define here (see below). They allow us to provide information about the underlying communication medium and the reliability of processes.

Rule (Init) initialises a session with n roles. Session initialisation introduces a fresh session channel and unguards the participants of the session. Finally, the message queues of this session are initialised with the empty list under the restriction of the session channel.

Rule (RSend) implements an asynchronous strongly reliable message transmission. As a result the value $\mathrm{eval}(y)$ is wrapped in a message and added to the end of the corresponding message queue and the continuation of the sender is unguarded. Rule (USend) is the

$$(\mathsf{Init}) \quad \overline{a}[\mathsf{n}](s).P_\mathsf{n} \mid \prod_{1 \leq \mathsf{i} \leq \mathsf{n}-1} a[\mathsf{i}](s).P_\mathsf{i} \longmapsto (\nu s)\left(\prod_{1 \leq \mathsf{i} \leq \mathsf{n}} P_\mathsf{i} \mid \prod_{1 \leq \mathsf{i},\mathsf{j} \leq \mathsf{n}, \mathsf{i} \neq \mathsf{j}} s_{\mathsf{i} \to \mathsf{j}}:[\,]\right)$$
$$\text{if } a \neq s$$

$$(\mathsf{RSend}) \quad s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{r}\langle y \rangle.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M}\#\langle v \rangle^\mathrm{r} \qquad \text{if } \mathrm{eval}(y) = v$$

$$(\mathsf{RGet}) \quad s[\mathsf{r}_1,\mathsf{r}_2]?_\mathrm{r}(x).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:\langle v \rangle^\mathrm{r}\#\mathrm{M} \longmapsto P\{v/x\} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{M}$$

$$(\mathsf{USend}) \quad s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{u}l\langle y \rangle.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M}\#l\langle v \rangle^\mathrm{u} \qquad \text{if } \mathrm{eval}(y) = v$$

$$(\mathsf{UGet}) \quad s[\mathsf{r}_1,\mathsf{r}_2]?_\mathrm{u}l\langle dv \rangle(x).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:l'\langle v \rangle^\mathrm{u}\#\mathrm{M} \longmapsto P\{v/x\} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{M}$$
$$\text{if } l \doteq l', \mathrm{FP}_{\mathtt{uget}}(s,\mathsf{r}_1,\mathsf{r}_2,l',\ldots)$$

$$(\mathsf{USkip}) \quad s[\mathsf{r}_1,\mathsf{r}_2]?_\mathrm{u}l\langle dv \rangle(x).P \longmapsto P\{dv/x\} \qquad \text{if } \mathrm{FP}_{\mathtt{uskip}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots)$$

$$(\mathsf{ML}) \quad s_{\mathsf{r}_1 \to \mathsf{r}_2}:l\langle v \rangle^\mathrm{u}\#\mathrm{M} \longmapsto s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M} \qquad \text{if } \mathrm{FP}_{\mathtt{ml}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots)$$

$$(\mathsf{RSel}) \quad s[\mathsf{r}_1,\mathsf{r}_2]!_\mathrm{r}l.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{M}\#l^\mathrm{r}$$

$$(\mathsf{RBran}) \quad s[\mathsf{r}_1,\mathsf{r}_2]?_\mathrm{r}\{l_i.P_i\}_{i \in \mathrm{I}} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:l^\mathrm{r}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{M} \qquad \text{if } l \doteq l_j, \, j \in \mathrm{I}$$

$$(\mathsf{WSel}) \quad s[\mathsf{r},\mathsf{R}]!_\mathrm{w}l.P \mid \prod_{\mathsf{r}_i \in \mathsf{R}} s_{\mathsf{r} \to \mathsf{r}_i}:\mathrm{M}_i \longmapsto P \mid \prod_{\mathsf{r}_i \in \mathsf{R}} s_{\mathsf{r} \to \mathsf{r}_i}:\mathrm{M}_i\#l^\mathrm{w}$$

$$(\mathsf{WBran}) \quad s[\mathsf{r}_1,\mathsf{r}_2]?_\mathrm{w}\{l_i.P_i\}_{i \in \mathrm{I},l_\mathrm{d}} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:l^\mathrm{w}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{M} \qquad \text{if } l \doteq l_j, \, j \in \mathrm{I}$$

$$(\mathsf{WSkip}) \quad s[\mathsf{r}_1,\mathsf{r}_2]?_\mathrm{w}\{l_i.P_i\}_{i \in \mathrm{I},l_\mathrm{d}} \longmapsto P_\mathrm{d} \qquad \text{if } \mathrm{FP}_{\mathtt{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2,\ldots)$$

$$(\mathsf{Crash}) \quad P \longmapsto \bot \qquad \text{if } \mathrm{FP}_{\mathtt{crash}}(P,\ldots)$$

Figure 3: Reduction Rules ($\longmapsto$) of Fault-Tolerant Processes (Part I).

counterpart of (RSend) for unreliable senders. (RGet) consumes a message that is marked as strongly reliable with the index r from the head of the respective message queue and replaces in the unguarded continuation of the receiver the bound variable x by the received value y.

There are two rules for the reception of a message in an unreliable communication that are guided by failure patterns. Rule (UGet) is similar to Rule (RGet), but specifies a failure pattern $\mathrm{FP}_{\mathtt{uget}}$ to decide whether this step is allowed. This failure pattern could, e.g., be used to reject messages that are too old. Moreover, $l \doteq l'$ is required to enforce that the static information in the transmitted label matches the expectation specified in the label of the receiver. As explained in Section 3.3, this allows to avoid communication mismatches. The Rule (USkip) allows to skip the reception of a message in an unreliable communication using a failure pattern $\mathrm{FP}_{\mathtt{uskip}}$ and instead substitutes the bound variable x in the continuation with the default value dv. The failure pattern $\mathrm{FP}_{\mathtt{uskip}}$ tells us whether a reception can be skipped (e.g. via failure detector).

Rule (RSel) puts the label l selected by r_1 at the end of the message queue towards r_2. Its weakly reliable counterpart (WSel) is similar, but puts the label at the end of all relevant message queues. With (RBran) a label is consumed from the top of a message queue and the receiver moves to the indicated branch. There are again two weakly reliable counterparts of (RBran). Rule (WBran) is similar to (RBran), whereas (WSkip) allows r_1 to skip the message and to move to its default branch if the failure pattern $\mathrm{FP}_{\mathtt{wskip}}$ holds. The requirement $l \doteq l_j$ in RBran and WBran ensures as usual that indeed the branch specified by the message at the queue is picked by the receiver. Note that this branch has to be identified by the statically available information in the respective labels.

The Rules (Crash) for *crash failures* and (ML) for *message loss*, describe failures of a system. With Rule (Crash) P can crash if $\mathrm{FP}_{\mathtt{crash}}$, where $\mathrm{FP}_{\mathtt{crash}}$ can e.g. model immortal processes or global bounds on the number of crashes. (ML) allows to drop an unreliable

(If-T)   `if` e `then` P `else` $P' \longmapsto P$ if e is true

(If-F)   `if` e `then` P `else` $P' \longmapsto P'$ if e is false

(Deleg) $s[\mathsf{r}_1, \mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \longmapsto P \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#s'[\mathsf{r}]$

(SRecv) $s[\mathsf{r}_1, \mathsf{r}_2]?((s'[\mathsf{r}])).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}s''[\mathsf{r}']\#\mathrm{M} \longmapsto P\{s''/s'\}\{\mathsf{r}'/\mathsf{r}\} \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}$

(Par) $P_1 \mid P_2 \longmapsto P_1' \mid P_2$ if $P_1 \longmapsto P_1'$

(Res) $(\nu x)P \longmapsto (\nu x)P'$ if $P \longmapsto P'$

(Rec) $(\mu X)P \longmapsto P\{(\mu X)P/X\}$

(Struc) $P_1 \longmapsto P_1'$ if $P_1 \equiv P_2,\ P_2 \longmapsto P_2',\ P_2' \equiv P_1'$

Figure 4: Reduction Rules ($\longmapsto$) of Fault-Tolerant Processes (Part II).

message if the failure pattern $\mathsf{FP_{ml}}$ is valid. $\mathsf{FP_{ml}}$ allows, e.g., to implement safe channels that never lose messages or a global bound on the number of lost messages.

Figure 4 provides the remaining reduction rules for conditionals, delegation, parallel composition, restriction, recursion, and structural congruence. They are standard.

Consider the implementation of $G_{\mathrm{dice,u}}$ in (2), i.e., an infinite variant of the dice game, where the players 1 and 2 use their respective last known sum x_{i} of former dice rolls as default value:

$$P_{\mathrm{dice,u}} = P_{3,\mathrm{u}} \mid P_{\mathsf{i},\mathrm{u}} \mid P_{2,\mathrm{u}}$$
$$P_{3,\mathrm{u}} = \overline{a}[3](s).(\mu X)s[3,1]!_{\mathrm{u}}\,roll\langle\mathsf{roll}(x_1)\rangle.s[3,2]!_{\mathrm{u}}\,roll\langle\mathsf{roll}(x_2)\rangle.X$$
$$P_{\mathsf{i},\mathrm{u}} = a[\mathsf{i}](s).(\mu X)s[\mathsf{i},3]?_{\mathrm{u}}\,roll\langle x_{\mathsf{i}}\rangle(x_{\mathsf{i}}).X$$

An unreliable communication in a global type specifies a communication that, due to system failures, may or may not happen. Moreover, regardless of the successful completion of this unreliable communication, the future behaviour of a well-typed system will follow its specification in the global type. Since the players 1 and 2 repeat the same kind of unreliable action, they may lose track of the current round. If they successfully receive a new sum of dice rolls from 3 they cannot be sure on how often 3 actually did roll the dice. Because of lost messages, they may have missed some former announcements of 3 and, because of their ability to skip the reception of messages, they may have proceeded to the next round before 3 rolled a dice. Because the information about the current round is irrelevant for the communication structure in this case, there is no need to enforce round information.

We deliberately do not specify failure pattern, although we usually assume that the failure patterns $\mathsf{FP_{uget}}$, $\mathsf{FP_{uskip}}$, and $\mathsf{FP_{wskip}}$ use only local information, whereas $\mathsf{FP_{ml}}$ and $\mathsf{FP_{crash}}$ may use global information of the system in the current run. We provide these predicates to allow for the implementation of system requirements or abstractions like failure detectors that are typical for distributed algorithms. Directly including them in the semantics has the advantage that all traces satisfy the corresponding requirements, i.e., all traces are valid w.r.t. the assumed system requirements. An example for the instantiation of these patterns is given implicitly via the Conditions 5.2.1–5.2.6 in Section 5 and explicitly in Section 6. If we instantiate the patterns $\mathsf{FP_{uget}}$ with true and the patterns $\mathsf{FP_{uskip}}$, $\mathsf{FP_{wskip}}$, $\mathsf{FP_{crash}}$, $\mathsf{FP_{ml}}$ with false, then we obtain a system without failures. In contrast, the instantiation of all five patterns with true results in a system where failures can happen completely non-deterministically at any time.

Note that we keep the failure patterns abstract and do not model how to check them in producing runs. Indeed system requirements such as bounds on the number of processes that can crash usually cannot be checked, but result from observations, i.e., system designers ensure that a violation of this bound is very unlikely and algorithm designers are willing to ignore these unlikely events. In particular, $\mathtt{FP}_{\mathtt{ml}}$ and $\mathtt{FP}_{\mathtt{crash}}$ are thus often implemented as oracles for verification, whereas e.g. $\mathtt{FP}_{\mathtt{uskip}}$ and $\mathtt{FP}_{\mathtt{wskip}}$ are often implemented by system specific time-outs. Note that we are talking about implementing these failure patterns and not formalising them. Failure patterns are abstractions of real world system requirements or software. We implement them by conditions providing the necessary guarantees that we need in general (i.e., for subject reduction and progress) or for the verification of concrete algorithms. In practice, we expect that the systems on which the verified algorithms are running satisfy the respective conditions. Accordingly, the session channels, roles, labels, and processes mentioned in Figure 3 are not parameters of the failure patterns, but just a vehicle to more formally specify the conditions on failure patterns in Section 5. An implementation may or may not use these information to implement these patterns but may also use other information such as runtime information about time or the number of processes, as indicated by the … in failure patterns in Figure 3 such as $\mathtt{FP}_{\mathtt{crash}}(P,\ldots)$.

Similarly, strongly reliable and weakly reliable interactions in potentially faulty systems are abstractions. They are usually implemented by handshakes and redundancy; replicated servers against crash failures and retransmission of late messages against message loss. Algorithm designers have to be aware of the additional costs of these interactions.

5. Typing Fault-Tolerant Processes

The type of processes is checked using typing rules that define the derivation of type judgments. Within type judgements, the type information are stored in type environments.

Definition 5.1 (Type Environments). The *global* and *session environments* are given by

$$\Gamma ::= \emptyset \mid \Gamma \cdot x{:}\mathrm{S} \mid \Gamma \cdot a{:}G \mid \Gamma \cdot l{:}\mathrm{S} \mid \Gamma \cdot X{:}t$$
$$\Delta ::= \emptyset \mid \Delta \cdot s[\mathsf{r}]{:}T \mid \Delta \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}^*$$

Assignments $x{:}\mathrm{S}$ of values to sorts are used to check whether transmitted values are well-sorted, i.e., sender and receiver expect the same sort. Assignments $a{:}G$ capture the global type of a session for session initialisation via the shared channel a. Assignments $l{:}\mathrm{S}$ link labels to sorts. Assignments $X{:}t$ of process variables to type variables are used to check the type of recursive processes.

Assignments $s[\mathsf{r}]{:}T$ of actors to local types are used to compare the behaviour of a process that implements this actor with its local specification T. Assignments $s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}^*$ allow to check the current content of a message queue $s_{\mathsf{r}_1 \to \mathsf{r}_2}$ against a list of message types MT.

We write $x \sharp \Gamma$ and $x \sharp \Delta$ if the name x does not occur in Γ and Δ, respectively. We use $\cdot$ to add an assignment provided that the new assignment is not in conflict with the type environment. More precisely, $\Gamma \cdot x{:}\mathrm{S}$ implies $x \sharp \Gamma$, $\Gamma \cdot l{:}\mathrm{S}$ implies $l \sharp \Gamma$, and $\Gamma \cdot X{:}s[\mathsf{r}]{:}t$ implies $X, t \sharp \Gamma$. Moreover, $\Delta \cdot s[\mathsf{r}]{:}T$ implies $(\nexists T'.\ s[\mathsf{r}]{:}T' \in \Delta)$ and $\Delta \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathcal{M}$ implies $(\nexists \mathcal{M}'.\ s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathcal{M}' \in \Delta)$. We naturally extend this operator towards sets, i.e., $\Gamma \cdot \Gamma'$ implies $(\forall A \in \Gamma'.\ \Gamma \cdot A)$ and $\Delta \cdot \Delta'$ implies $(\forall A \in \Delta'.\ \Delta \cdot A)$. The conditions described for the operator $\cdot$ for global and session environments are referred to as *linearity*. Accordingly, we

$$(\text{Req})\ \frac{a{:}G \in \Gamma \quad |\mathrm{R}(G)| = \mathsf{n} \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{n}]{:}G{\upharpoonright}_{\mathsf{n}}}{\Gamma \vdash \overline{a}[\mathsf{n}](s).P \triangleright \Delta}$$

$$(\text{Acc})\ \frac{a{:}G \in \Gamma \quad 0 < \mathsf{r} < |\mathrm{R}(G)| \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}G{\upharpoonright}_{\mathsf{r}}}{\Gamma \vdash a[\mathsf{r}](s).P \triangleright \Delta}$$

$$(\text{RSend})\ \frac{\Gamma \Vdash y{:}\mathrm{S} \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}\langle y\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T}$$

$$(\text{RGet})\ \frac{x\sharp(\Gamma,\Delta,s) \quad \Gamma \cdot x{:}\mathrm{S} \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{r}}(x).P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{r}}\langle \mathrm{S}\rangle.T}$$

$$(\text{USend})\ \frac{\Gamma \Vdash y{:}\mathrm{S} \quad l \doteq l' \quad l'{:}\mathrm{S} \in \Gamma \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{u}}l\langle y\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{u}}l'\langle \mathrm{S}\rangle.T}$$

$$(\text{UGet})\ \frac{x\sharp(\Gamma,\Delta,s) \quad \Gamma \Vdash v{:}\mathrm{S} \quad l \doteq l' \quad l'{:}\mathrm{S} \in \Gamma \quad \Gamma \cdot x{:}\mathrm{S} \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{u}}l\langle v\rangle(x).P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{u}}l'\langle \mathrm{S}\rangle.T}$$

$$(\text{RSel})\ \frac{j \in \mathrm{I} \quad l \doteq l_j \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}l.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i\in\mathrm{I}}}$$

$$(\text{RBran})\ \frac{\forall j \in \mathrm{I}_2.\ \exists i \in \mathrm{I}_1.\ l_i \doteq l'_j \wedge \Gamma \vdash P_i \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{r}}\{l_i.P_i\}_{i\in\mathrm{I}_1} \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{r}}\{l'_i.T_i\}_{i\in\mathrm{I}_2}}$$

$$(\text{WSel})\ \frac{j \in \mathrm{I} \quad l \doteq l_j \quad \Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}T_j}{\Gamma \vdash s[\mathsf{r},\mathsf{R}]!_{\mathrm{w}}l.P \triangleright \Delta \cdot s[\mathsf{r}]{:}[\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i\in\mathrm{I}}}$$

$$(\text{WBran})\ \frac{l_{\mathrm{d}} \doteq l'_{\mathrm{d}} \quad \forall j \in \mathrm{I}_2.\ \exists i \in \mathrm{I}_1.\ l_i \doteq l'_j \wedge \Gamma \vdash P_i \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i\in\mathrm{I}_1,l_{\mathrm{d}}} \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{w}}\{l'_i.T_i\}_{i\in\mathrm{I}_2,l'_{\mathrm{d}}}}$$

$$(\text{Deleg})\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s'[\mathsf{r}]{:}T'}$$

$$(\text{SRecv})\ \frac{\Gamma \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T \cdot s'[\mathsf{r}]{:}T'}{\Gamma \vdash s[\mathsf{r}_1,\mathsf{r}_2]?(\!(s'[\mathsf{r}])\!).P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?\langle s'[\mathsf{r}]{:}T'\rangle.T}$$

$$(\text{End})\ \frac{}{\Gamma \vdash \mathbf{0} \triangleright \emptyset} \qquad (\text{Crash})\ \frac{\mathrm{nsr}(\Delta)}{\Gamma \vdash \bot \triangleright \Delta} \qquad (\text{If})\ \frac{\Gamma \Vdash e{:}\mathbb{B} \quad \Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \triangleright \Delta}$$

$$(\text{Par})\ \frac{\Gamma \vdash P \triangleright \Delta \quad \Gamma \vdash P' \triangleright \Delta'}{\Gamma \vdash P \mid P' \triangleright \Delta \cdot \Delta'} \qquad (\text{Res1})\ \frac{x\sharp(\Gamma,\Delta) \quad \Gamma \cdot x{:}\mathrm{S} \vdash P \triangleright \Delta}{\Gamma \vdash (\nu x)P \triangleright \Delta}$$

$$(\text{Rec})\ \frac{\Gamma \cdot X{:}t \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}T}{\Gamma \vdash (\mu X)P \triangleright \Delta \cdot s[\mathsf{r}]{:}(\mu t)T} \qquad (\text{Var})\ \frac{}{\Gamma \cdot X{:}t \vdash X \triangleright s[\mathsf{r}]{:}t}$$

Figure 5: Typing Rules for Fault-Tolerant Systems.

denote type environments that satisfy these properties as *linear* and restrict in the following our attention to linear environments. We abstract in session environments from assignments towards terminated local types, i.e., $\Delta \cdot s[\mathsf{r}]{:}\mathtt{end} = \Delta$.

A *type judgement* is of the form $\Gamma \vdash P \rhd \Delta$, where Γ is a global environment, $P \in \mathcal{P}$ is a process, and Δ is a session environment. We use *typing rules* to derive type judgements, where we assume that all mentioned global types are well-formed and projectable, all local types are well-formed, and all environments are linear. A process P is *well-typed* w.r.t. Γ and Δ if $\Gamma \vdash P \rhd \Delta$ can be derived from the rules in the Figures 5 and 6. We write nsr(Δ) if none of the prefixes in T is strongly reliable or for delegation for all local types T in Δ and if Δ does not contain message queues. With $\Gamma \Vdash y$:S we check that y is an expression of the sort S if all names x in y are replaced by arbitrary values of sort S_x for x:$\mathrm{S}_x \in \Gamma$.

Let us consider the interaction cases in Figure 5. We observe that all new cases are quite similar to their strongly reliable counterparts.

Rule (RSend) checks strongly reliable senders, i.e., requires a matching strongly reliable sending in the local type of the actor and compares the actor with this type. With $\Gamma \Vdash y$:S we check that y is an expression of the sort S if all names x in y are replaced by arbitrary values of sort S_x for x:$\mathrm{S}_x \in \Gamma$. Then the continuation of the process is checked against the continuation of the type. The unreliable case is very similar, but additionally checks that the label is assigned to the sort of the expression in Γ. Rule (RGet) type strongly reliable receivers, where again the prefix is checked against a corresponding type prefix and the assumption x:S is added for the continuation. Again the unreliable case is very similar, but apart from the label also checks the sort of the default value.

Rule (RSel) checks the strongly reliable selection prefix, that the selected label matches one of the specified labels, and that the process continuation is well-typed w.r.t. the type continuation following the selected label. The only difference in the weakly reliable case is the set of roles for the receivers. For strongly reliable branching in (RBran) we check the prefix and that for each branch in the type there is a matching branch in the process that is well-typed w.r.t. the respective branch in the type. For the weakly reliable case we have to additionally check that the default labels of the process and the type coincide.

Rule (Crash) for crashed processes checks that nsr(Δ), i.e., that for every type G or T in Δ the predicate nsr(G) or nsr(T) holds.

Figure 6 presents the runtime typing rules, i.e., the typing rules for processes that may result from steps of a system that implements a global type. Since it covers only operators that are not part of initial systems, a type checking tool might ignore them. We need these rules however for the proofs of progress and subject reduction. Under the assumption that initial systems cannot contain crashed processes, Rule (Crash) may be moved to the set of runtime typing rules.

Rule (Res2) types sessions that are already initialised and that may have performed already some of the steps described by their global type. The relation $\stackrel{s}{\mapsto}$ is given in Figure 7 and describes how a session environment evolves alongside reductions of the system, i.e., it emulates the reduction steps of processes. As an example consider the rule $\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]!_{\mathsf{r}}\langle \mathrm{S}\rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT} \stackrel{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT}\#\langle \mathrm{S}\rangle^{\mathsf{r}}$ that emulates (RSend). Let $\stackrel{s}{\Mapsto}$ denote the reflexive and transitive closure of $\stackrel{s}{\mapsto}$.

We have to prove that our extended type system satisfies the standard properties of MPST, i.e., subject reduction and progress. Because of the failure pattern in the reduction semantics in Figure 3, subject reduction and progress do not hold in general. Instead we have to fix conditions on failure patterns that ensure these properties. Subject reduction needs one condition on crashed processes and progress requires that no part of the system is blocked. In fact, different instantiations of these failure patterns may allow for progress.

$$\text{(Res2)} \frac{\{s[\mathsf{r}]{:}G{\upharpoonright}_{\mathsf{r}} \mid \mathsf{r} \in \mathrm{R}(G)\} \cdot \{s_{\mathsf{r}\to\mathsf{r}'}{:}[\,] \mid \mathsf{r},\mathsf{r}' \in \mathrm{R}(G') \wedge \mathsf{r} \neq \mathsf{r}'\} \overset{s}{\mapsto} \Delta' \quad s\sharp(\Gamma,\Delta) \quad a{:}G \in \Gamma \quad \Gamma \vdash P \triangleright \Delta \cdot \Delta'}{\Gamma \vdash (\nu s)P \triangleright \Delta}$$

$$\text{(MQComR)} \frac{\Gamma \Vdash v{:}\mathrm{S} \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\langle v \rangle^{\mathrm{r}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\langle \mathrm{S} \rangle^{\mathrm{r}}\#\mathrm{MT}}$$

$$\text{(MQComU)} \frac{\Gamma \Vdash v{:}\mathrm{S} \quad l \doteq l' \quad l'{:}\mathrm{S} \in \Gamma \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l\langle v \rangle^{\mathrm{u}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l'\langle \mathrm{S} \rangle^{\mathrm{u}}\#\mathrm{MT}}$$

$$\text{(MQBranR)} \frac{l \doteq l' \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l^{\mathrm{r}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l'^{\mathrm{r}}\#\mathrm{MT}}$$

$$\text{(MQBranW)} \frac{l \doteq l' \quad \Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l^{\mathrm{w}}\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l'^{\mathrm{w}}\#\mathrm{MT}}$$

$$\text{(MQDeleg)} \frac{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}s'[\mathsf{r}]\#\mathrm{M} \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}s'[\mathsf{r}]\#\mathrm{MT}} \qquad \text{(MQNil)} \frac{}{\Gamma \vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}[\,] \triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}[\,]}$$

Figure 6: Runtime Typing Rules for Fault-Tolerant Systems.

We leave it for future work to determine what kind of conditions on failure patterns or requirements on their interactions are necessary. Here, we consider only one such set.

Condition 5.2 (Failure Pattern).
(1) If $\mathtt{FP_{crash}}(P,\ldots)$, then $\mathrm{nsr}(P)$.
(2) The failure pattern $\mathtt{FP_{uget}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots)$ is always valid.
(3) The pattern $\mathtt{FP_{ml}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots)$ is valid iff $\mathtt{FP_{uskip}}(s,\mathsf{r}_2,\mathsf{r}_1,l,\ldots)$ is valid.
(4) If $\mathtt{FP_{crash}}(P,\ldots)$ and $s[\mathsf{r}] \in \mathrm{A}(P)$ is an actor then eventually $\mathtt{FP_{uskip}}(s,\mathsf{r}_2,\mathsf{r},l,\ldots)$ and $\mathtt{FP_{wskip}}(s,\mathsf{r}_2,\mathsf{r},l,\ldots)$ for all r_2, l.
(5) If $\mathtt{FP_{crash}}(P,\ldots)$ and $s[\mathsf{r}] \in \mathrm{A}(P)$ then eventually $\mathtt{FP_{ml}}(s,\mathsf{r}_1,\mathsf{r},l,\ldots)$ for all r_1, l.
(6) If $\mathtt{FP_{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2,\ldots)$ then $s[\mathsf{r}_2]$ is crashed, i.e., the system does no longer contain an actor $s[\mathsf{r}_2]$ and the message queue $s_{\mathsf{r}_2\to\mathsf{r}_1}$ is empty.

The crash of a process should not block strongly reliable actions, i.e., only processes with $\mathrm{nsr}(P)$ can crash (Condition 5.2.1). Condition 5.2.2 requires that no process can refuse to consume a message on its queue to prevent deadlocks that may arise from refusing a message that is never dropped. Condition 5.2.3 requires that if a message can be dropped from a message queue then the corresponding receiver has to be able to skip this message and vice versa. Similarly, processes that wait for messages from a crashed process have to be able to skip (Condition 5.2.4) and all messages of a queue towards a crashed receiver can be dropped (Condition 5.2.5). Finally, weakly reliable branching requests should not be lost. To ensure that the receiver of such a branching request can proceed if the sender is crashed but is not allowed to skip the reception of the branching request before the sender crashed, we require that $\mathtt{FP_{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2,\ldots)$ is false as long as $s[\mathsf{r}_2]$ is alive or messages on the respective queue are still in transit (Condition 5.2.6).

The combination of the 6 conditions in Conditions 5.2 might appear quite restrictive as e.g. the combination of the Condition 5.2.4 and 5.2.6 ensures the correct behaviour of weakly reliable branching such that branching messages can be skipped if and only if the

$$(\mathsf{RSend}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathsf{S}\rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT}\#\langle \mathsf{S}\rangle^{\mathrm{r}}}$$

$$(\mathsf{RGet}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?_{\mathrm{r}}\langle \mathsf{S}\rangle.T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:\langle \mathsf{S}\rangle^{\mathrm{r}}\#\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{MT}}$$

$$(\mathsf{USend}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]!_{\mathrm{u}}l\langle \mathsf{S}\rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT}\#l\langle \mathsf{S}\rangle^{\mathrm{u}}}$$

$$(\mathsf{UGet}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?_{\mathrm{u}}l\langle \mathsf{S}\rangle.T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:l\langle \mathsf{S}\rangle^{\mathrm{u}}\#\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{MT}}$$

$$(\mathsf{USkip}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?_{\mathrm{u}}l\langle \mathsf{S}\rangle.T \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T}$$

$$(\mathsf{ML}) \; \frac{}{\Delta \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:l\langle \mathsf{S}\rangle^{\mathrm{u}}\#\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT}}$$

$$(\mathsf{RSel}) \; \frac{j \in \mathrm{I}}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i \in \mathrm{I}} \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T_j \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT}\#l_j^{\mathrm{r}}}$$

$$(\mathsf{RBran}) \; \frac{j \in \mathrm{I}}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?_{\mathrm{r}}\{l_i.T_i\}_{i \in \mathrm{I}} \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:l_j^{\mathrm{r}}\#\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T_j \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{MT}}$$

$$(\mathsf{WSel}) \; \frac{j \in \mathrm{I} \quad \mathsf{R} = \{\mathsf{r}_1, \dots, \mathsf{r}_n\}}{\begin{array}{c}\Delta \cdot s[\mathsf{r}]:[\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i \in \mathrm{I}, l_{\mathrm{d}}} \cdot s_{\mathsf{r} \to \mathsf{r}_1}:\mathrm{MT}_1 \cdot \dots \cdot s_{\mathsf{r} \to \mathsf{r}_n}:\mathrm{MT}_n \overset{s}{\mapsto} \\ \Delta \cdot s[\mathsf{r}]:T_j \cdot s_{\mathsf{r} \to \mathsf{r}_1}:\mathrm{MT}_1\#l_j^{\mathrm{w}} \cdot \dots \cdot s_{\mathsf{r} \to \mathsf{r}_n}:\mathrm{MT}_n\#l_j^{\mathrm{w}}\end{array}}$$

$$(\mathsf{WBran}) \; \frac{j \in \mathrm{I}}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?_{\mathrm{w}}\{l_i.T_i\}_{i \in \mathrm{I}, l_{\mathrm{d}}} \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:l_j^{\mathrm{w}}\#\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T_j \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{MT}}$$

$$(\mathsf{WSkip}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?_{\mathrm{w}}\{l_i.T_i\}_{i \in \mathrm{I}, l_{\mathrm{d}}} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T_{\mathrm{d}}}$$

$$(\mathsf{Rec}) \; \frac{}{\Delta \cdot s[\mathsf{r}]:(\mu t)T \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}]:t\{(\mu t)T/t\}}$$

$$(\mathsf{Deleg}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]!\langle s'[\mathsf{r}]:T'\rangle.T \cdot s'[\mathsf{r}]:T' \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}:\mathrm{MT}\#s'[\mathsf{r}]}$$

$$(\mathsf{SRecv}) \; \frac{}{\Delta \cdot s[\mathsf{r}_1]:[\mathsf{r}_2]?\langle s'[\mathsf{r}]:T'\rangle.T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:s'[\mathsf{r}]\#\mathrm{MT} \overset{s}{\mapsto} \Delta \cdot s[\mathsf{r}_1]:T \cdot s'[\mathsf{r}]:T' \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}:\mathrm{MT}}$$

Figure 7: Reduction Rules for Session Environments.

respective sender has crashed. An implementation of such a weakly reliable interaction in an asynchronous system that is subject to message losses and process crashes, might require something like a perfect failure detector or actually solving consensus[1]. It is important to remember that these conditions are minimal assumptions on the system requirements and that system requirements are abstractions. Parts of them may be realised by actual software-code (which then allows to check them), whereas other parts of the system requirements

---

[1]Note that we consider in Section 6 a consensus algorithm. So, if the Condition 5.2 requires a solution of consensus, an example on top of that solving consensus would be pointless.

may not be realised at all but rather observed (which then does not allow to verify them). Weakly reliable branching is a good example of this case. The easiest way to obtain a weakly reliable interaction, is by using a handshake communication and time-outs. If the sender time-outs while waiting for an acknowledgement, it resends the message. If the sender does not hear from its receiver for a long enough period of time, it assumes that the receiver has crashed and proceeds. With carefully chosen time-frames for the time-outs, this approach is a compromise between correctness and efficiency. In a theoretical sense, it is clearly not correct. There is no time-frame such that the sender can be really sure that the receiver has crashed. From a practical point of view, this is not so problematic, since in many systems failures are very unlikely. If failures that are so severe that they are not captured by the time-outs are extremely unlikely, then it is often much more efficient to just accept that the algorithm is not correct in these cases. Trying to obtain an algorithm that is always correct might be impossible or at least usually results into very inefficient algorithms. Moreover, verifying this requires to also verify the underlying communication infrastructure and the way in that failures may occur, which is impossible or at least impracticable. Because of that, it is an established method to verify the correctness of algorithms w.r.t. given system requirements (e.g. in [CT96, Lam01, vST17]), even if these system requirements are not verified and often do not hold in all (but only nearly all) cases.

Let us have a closer look at the typing rules in the Figures 5 and 6. We observe that all typing rules are clearly distinguished by the outermost operator of the process in the conclusion except that there are two typing rules for restriction. With that, given a type judgement $\Gamma \vdash P \rhd \Delta$, we can use the structure of P—with a case split for restriction—to reason about the structure of the proof tree that was necessary to obtain $\Gamma \vdash P \rhd \Delta$ and from that derive conditions about the nature of the involved type environments. If P is e.g. a parallel composition $P_1 \mid P_2$ then, since there is only one rule to type parallel compositions (the Rule (Par)), $\Gamma \vdash P_1 \mid P_2 \rhd \Delta$ implies that there are Δ_1, Δ_2 such that $\Delta = \Delta_1 \cdot \Delta_2$, $\Gamma \vdash P_1 \rhd \Delta_1$, and $\Gamma \vdash P_2 \rhd \Delta_2$. In the following, we write 'by Rule (Par)' as short hand for 'by the clear distinction of the typing rules by the process in the conclusion and Rule (Par) in particular' and similar for the other rules.

In the following we prove some properties of our MPST variant. We start with an auxiliary result, proving that structural congruence preserves the validity of type judgements. The proof is by induction on $P \equiv P'$. In each case we can use the information about the structure of the process that is provided by the considered rule of structural congruence to conclude on the last few typing rules that had to be applied to derive the type judgement in the assumption. From these partial proof trees we obtain enough information to construct the proof tree for the conclusion.

Lemma 5.3 (Subject Congruence). *If $\Gamma \vdash P \rhd \Delta$ and $P \equiv P'$ then $\Gamma \vdash P' \rhd \Delta$.*

Proof. The proof is by induction on $P \equiv P'$.

Case $P \mid \mathbf{0} \equiv P$: Assume $\Gamma \vdash P \mid \mathbf{0} \rhd \Delta$. By the Rule (Par), then there are $\Delta_P, \Delta_{\mathbf{0}}$ such that $\Delta = \Delta_P \cdot \Delta_{\mathbf{0}}$, $\Gamma \vdash P \rhd \Delta_P$, and $\Delta \vdash \mathbf{0} \rhd \Delta_{\mathbf{0}}$. Moreover, by Rule (End), $\Gamma \vdash \mathbf{0} \rhd \Delta_{\mathbf{0}}$ implies that $\Delta_{\mathbf{0}} = \emptyset$ and, thus, $\Delta = \Delta_P$. Then also $\Gamma \vdash P \rhd \Delta$.

For the opposite direction assume $\Gamma \vdash P \rhd \Delta$. By Rule (End), $\Gamma \vdash \mathbf{0} \rhd \emptyset$. With Rule (Par) and because $\Delta = \Delta \cdot \emptyset$, then $\Gamma \vdash P \mid \mathbf{0} \rhd \Delta$.

Case $P_1 \mid P_2 \equiv P_2 \mid P_1$: Assume $\Gamma \vdash P_1 \mid P_2 \rhd \Delta$. By Rule (Par), then there are $\Delta_{P_1}, \Delta_{P_2}$ such that $\Delta = \Delta_{P_1} \cdot \Delta_{P_2}$ and $\Gamma \vdash P_i \rhd \Delta_{P_i}$. By Rule (Par) and since $\Delta = \Delta_{P_1} \cdot \Delta_{P_2}$

implies $\Delta = \Delta_{P_2} \cdot \Delta_{P_1}$, then $\Gamma \vdash P_2 \mid P_1 \triangleright \Delta$.
The opposite direction is similar.

Case $P_1 \mid (P_2 \mid P_3) \equiv (P_1 \mid P_2) \mid P_3$**:** Assume $\Gamma \vdash P_1 \mid (P_2 \mid P_3) \triangleright \Delta$.
By Rule (Par), then there are $\Delta_{P_1}, \Delta_{P_2}, \Delta_{P_3}$ such that $\Delta = \Delta_{P_1} \cdot (\Delta_{P_2} \cdot \Delta_{P_3})$ and
$\Gamma \vdash P_i \triangleright \Delta_{P_i}$. By Rule (Par) and because $\Delta = (\Delta_{P_1} \cdot \Delta_{P_2}) \cdot \Delta_{P_3}$, then $\Gamma \vdash (P_1 \mid P_2) \mid$
$P_3 \triangleright \Delta$.
The opposite direction is similar.

Case $(\mu X)\mathbf{0} \equiv \mathbf{0}$**:** Assume $\Gamma \vdash (\mu X)\mathbf{0} \triangleright \Delta$. By Rule (Rec), then $\Delta = \Delta' \cdot s[\mathsf{r}]{:}(\mu t)T$,
$\Gamma \cdot X{:}t \vdash \mathbf{0} \triangleright \Delta' \cdot s[\mathsf{r}]{:}T$, and, by Rule (End), then $\Delta' = \emptyset$ and $T = \mathsf{end}$. Since
$(\mu t)\mathsf{end} = \mathsf{end}$ and $\Delta \cdot s[\mathsf{r}]{:}\mathsf{end} = \Delta$, then $\Delta = \emptyset$. By Rule (End), then $\Gamma \vdash \mathbf{0} \triangleright \Delta$.
For the opposite direction assume $\Gamma \vdash \mathbf{0} \triangleright \Delta$. By Rule (End), then $\Delta = \emptyset$. By
Rule (End), then also $\Gamma \cdot X{:}t \vdash \mathbf{0} \triangleright \Delta$. By Rule (Rec) and since $(\mu t)\mathsf{end} = \mathsf{end}$ and
$\Delta \cdot s[\mathsf{r}]{:}\mathsf{end} = \Delta$, then $\Gamma \vdash (\mu X)\mathbf{0} \triangleright \Delta$.

Case $(\nu x)\mathbf{0} \equiv \mathbf{0}$**:** Assume $\Gamma \vdash (\nu x)\mathbf{0} \triangleright \Delta$. By one of the Rules (Res1) or (Res2), then there
are Γ', Δ' such that $\Gamma \cdot \Gamma' \vdash \mathbf{0} \triangleright \Delta$ or $\Gamma \cdot \Gamma' \vdash \mathbf{0} \triangleright \Delta \cdot \Delta'$. In both cases we can conclude
with Rule (End) that the session environment is empty, i.e., $\Delta = \emptyset$ and $\Delta \cdot \Delta' = \emptyset$. By
Rule (End), then $\Gamma \vdash \mathbf{0} \triangleright \Delta$.
For the opposite direction assume $\Gamma \vdash \mathbf{0} \triangleright \Delta$. By Rule (End), then $\Delta = \emptyset$. By
Rule (End), then $\Gamma \cdot x{:}\mathrm{S} \vdash \mathbf{0} \triangleright \Delta$ for some sort S—regardless of whether x is a value or
a session channel. By Rule (Res1), then $\Gamma \vdash (\nu x)\mathbf{0} \triangleright \Delta$.

Case $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$**:** Assume $\Gamma \vdash (\nu x)(\nu y)P \triangleright \Delta$. By alpha-conversion and the
Rules (Res1) and (Res2), then there is Γ' and some (possibly empty) Δ' such that—for
all combinations of the Rules (Res1) and (Res2) for the restrictions of x and y—we have
$\Gamma \cdot \Gamma' \vdash P \triangleright \Delta \cdot \Delta'$. By the commutativity and associativity of $\cdot$ and two corresponding
applications of the Rules (Res1) and (Res2), then also $\Gamma \vdash (\nu y)(\nu x)P \triangleright \Delta$.
The opposite direction is similar.

Case $(\nu x)(P_1 \mid P_2) \equiv P_1 \mid (\nu x)P_2$ **if** $x \notin \mathrm{FN}(P_1)$**:** Assume $\Gamma \vdash (\nu x)(P_1 \mid P_2) \triangleright \Delta$. By one
of the Rules (Res1) or (Res2), then $x \sharp (\Gamma, \Delta)$ and there are Γ', Δ' such that $\Gamma \cdot \Gamma' \vdash P_1 \mid$
$P_2 \triangleright \Delta \cdot \Delta'$, where Γ' assigns to x either a sort or a global type and Δ' is either empty or
contains only actors and message queues. Since $x \notin \mathrm{FN}(P_1)$ and by Rule (Par), then
there are $\Delta_{P_1}, \Delta_{P_2}$ such that $\Delta = \Delta_{P_1} \cdot \Delta_{P_2}$, $\Gamma \vdash P_1 \triangleright \Delta_{P_1}$, and $\Gamma \cdot \Gamma' \vdash P_2 \triangleright \Delta_{P_2} \cdot \Delta'$.
With one of the Rules (Res1) or (Res2), then $\Gamma \vdash (\nu x)P_2 \triangleright \Delta_{P_2}$. With Rule (Par), then
$\Gamma \vdash P_1 \mid (\nu x)P_2 \triangleright \Delta$.
The opposite direction is similar. $\square$

Moreover, types are preserved modulo substitution of names by values of the same sort.
The proof is by induction on the typing rules.

Lemma 5.4 (Substitution)**.** *If* $\Gamma \cdot c{:}\mathrm{S}_c \vdash Q \triangleright \Delta$ *and* $\Gamma \Vdash d{:}\mathrm{S}_c$, *then* $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

Proof. The proof is by induction on the derivation of $\Gamma \cdot c{:}\mathrm{S}_c \vdash Q \triangleright \Delta$.

(Req): Then, $Q = \overline{a}[\mathsf{n}](s).P$, $a{:}G \in \Gamma$, $|\mathrm{R}(G)| = \mathsf{n}$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \triangleright \Delta \cdot s[\mathsf{n}]{:}G{\upharpoonright}_\mathsf{n}$.
Without loss of generality, assume $s \neq c$. Because of linearity, $a{:}G \in \Gamma$ implies
$a \neq c$. By the induction hypothesis, $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \triangleright \Delta \cdot s[\mathsf{n}]{:}G{\upharpoonright}_\mathsf{n}$ and $\Gamma \Vdash d{:}\mathrm{S}_c$ imply
$\Gamma \vdash P\{d/c\} \triangleright \Delta \cdot s[\mathsf{n}]{:}G{\upharpoonright}_\mathsf{n}$. By (RReq), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(Acc): Then, $Q = a[\mathsf{r}](s).P$, $a{:}G \in \Gamma$, $0 < \mathsf{r} < |\mathrm{R}(G)|$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}G{\upharpoonright}_\mathsf{r}$.
Without loss of generality, assume $s \neq c$. Because of linearity, $a{:}G \in \Gamma$ implies $a \neq c$.
By the induction hypothesis, $\Gamma \vdash P\{d/c\} \triangleright \Delta \cdot s[\mathsf{r}]{:}G{\upharpoonright}_\mathsf{r}$. By (Acc), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(RSend): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}}\langle y\rangle.P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T$, $\Gamma \cdot c{:}\mathrm{S}_c \Vdash y{:}\mathrm{S}$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. Because $\Gamma \cdot c{:}\mathrm{S}_c \Vdash y{:}\mathrm{S}$, $s \neq c$. With $\Gamma \Vdash d{:}\mathrm{S}_c$, then $\Gamma \Vdash y\{d/c\}{:}\mathrm{S}$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. By (RSend), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(RGet): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}(x).P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{r}}\langle \mathrm{S}\rangle.T$, $x\sharp(\Gamma, c, \Delta', s)$, and $\Gamma \cdot c{:}\mathrm{S}_c \cdot x{:}\mathrm{S} \vdash P_i \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. Because $\Gamma \cdot c{:}\mathrm{S}_c \cdot x{:}\mathrm{S} \vdash P_i \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$, $s \neq c$. By the induction hypothesis, $\Gamma \cdot x{:}\mathrm{S} \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. By (RGet), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(USend): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{u}}l\langle y\rangle.P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{u}}l'\langle \mathrm{S}\rangle.T$, $l \doteq l'$, $\Gamma \cdot c{:}\mathrm{S}_c \Vdash y{:}\mathrm{S}$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. Because of $\Gamma \cdot c{:}\mathrm{S}_c \Vdash y{:}\mathrm{S}$, $s \neq c$. With $\Gamma \Vdash d{:}\mathrm{S}_c$, then $\Gamma \Vdash y\{d/c\}{:}\mathrm{S}$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. By (USend), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(UGet): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{u}}l\langle v\rangle(x).P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{u}}l'\langle \mathrm{S}\rangle.T$, $l \doteq l'$, $\Gamma \cdot c{:}\mathrm{S}_c \Vdash v{:}\mathrm{S}$, $x\sharp(\Gamma, c, \Delta', s)$, and $\Gamma \cdot c{:}\mathrm{S}_c \cdot x{:}\mathrm{S} \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. Because of $\Gamma \cdot c{:}\mathrm{S}_c \Vdash v{:}\mathrm{S}$, $s \neq c$. With $\Gamma \Vdash d{:}\mathrm{S}_c \in \Gamma$, then $\Gamma \Vdash v\{d/c\}{:}\mathrm{S}$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. By (UGet), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(RSel): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]!_{\mathrm{r}}l.P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i \in \mathrm{I}}$, $j \in \mathrm{I}$, $l \doteq l_j$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$. By (RSel), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(RBran): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{r}}\{l_i.P_i\}_{i \in \mathrm{I}_1}$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{r}}\{l_i.T_i\}_{i \in \mathrm{I}_2}$, and, for all $j \in \mathrm{I}_2$ exists some $i \in \mathrm{I}_1$ such that $l_i \doteq l_j$ and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P_i \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$. Fix j and i. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash P_i \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash P_i\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$. By (RBran), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(WSel): Then, $Q = s[\mathsf{r}, \mathsf{R}]!_{\mathrm{w}}l.P$, $\Delta = \Delta' \cdot s[\mathsf{r}]{:}[\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i \in \mathrm{I},l_{\mathrm{d}}}$, $j \in \mathrm{I}$, $l \doteq l_j$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}]{:}T_j$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}]{:}T_j$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}]{:}T_j$. By (WSel), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(WBran): Then, $Q = s[\mathsf{r}_1, \mathsf{r}_2]?_{\mathrm{w}}\{l_i.P_i\}_{i \in \mathrm{I}_1,l_{\mathrm{d}}}$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathrm{w}}\{l_i.T_i\}_{i \in \mathrm{I}_2,l'_{\mathrm{d}}}$, $l_{\mathrm{d}} \doteq l'_{\mathrm{d}}$, and, for all $j \in \mathrm{I}_2$ exists some $i \in \mathrm{I}_1$ such that $l_i \doteq l_j$ and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P_i \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$. Fix j and i. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash P_i \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash P_i\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T_j$. By (WBran), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(If): Then, $Q = \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P'$, $\Gamma \cdot c{:}\mathrm{S}_c \Vdash e{:}\mathbb{B}$, $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P' \rhd \Delta$. With $\Gamma \Vdash d{:}\mathrm{S}_c \in \Gamma$, then $\Gamma \Vdash e\{d/c\}{:}\mathbb{B}$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta$ and $\Gamma \vdash P'\{d/c\} \rhd \Delta$. By (If), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(Deleg): Then $Q = s[\mathsf{r}_1, \mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s'[\mathsf{r}]{:}T'$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash Q \rhd \Delta$, $c \neq s$ and $c \neq s'$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T$. By (Deleg), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(SRecv): Then $Q = s[\mathsf{r}_1, \mathsf{r}_2]?(\!(s'[\mathsf{r}])\!).P$, $\Delta = \Delta' \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?\langle s'[\mathsf{r}]{:}T'\rangle.T$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T \cdot s'[\mathsf{r}]{:}T'$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T \cdot s'[\mathsf{r}]{:}T'$, $c \neq s$ and $c \neq s'$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}_1]{:}T \cdot s'[\mathsf{r}]{:}T'$. By (SRecv), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(Par): Then, $Q = P \mid P'$, $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \rhd \Delta$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P' \rhd \Delta$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \rhd \Delta$ and $\Gamma \vdash P'\{d/c\} \rhd \Delta$. By (Par), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(Res1): Then, $Q = (\nu x)P$, $x\sharp(\Gamma, c, \Delta)$, and $\Gamma \cdot c{:}\mathrm{S}_c \cdot x{:}\mathrm{S} \vdash P \rhd \Delta$. By the induction hypothesis, $\Gamma \cdot x{:}\mathrm{S} \vdash P\{d/c\} \rhd \Delta$. By (Res1), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(Rec): Then, $Q = (\mu X)P$, $\Delta = \Delta' \cdot s[\mathsf{r}]{:}T$, and $\Gamma \cdot c{:}\mathrm{S}_c \cdot X{:}t \vdash P \rhd \Delta' \cdot s[\mathsf{r}]{:}T$. By the induction hypothesis, $\Gamma \cdot X{:}t \vdash P\{d/c\} \rhd \Delta' \cdot s[\mathsf{r}]{:}T$. By (Rec), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(Var): Then, $Q = X$, $\Gamma = \Gamma' \cdot X{:}s[\mathsf{r}]{:}t$, and $\Delta = s[\mathsf{r}]{:}t$. By (Var), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(End): Then, $Q = \mathbf{0}$ and $\Delta = \emptyset$. By (End), then $\Gamma \vdash Q\{d/c\} \rhd \Delta$.

(Crash): Then, $Q = \bot$ and $\mathrm{nsr}(\Delta)$. By (UCrash), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(Res2): Then, $\{s[\mathsf{r}]{:}G\!\restriction_\mathsf{r} \mid \mathsf{r} \in \mathrm{R}(G)\} \cdot \{s_{\mathsf{r} \to \mathsf{r}'}{:}[\,] \mid \mathsf{r}, \mathsf{r}' \in \mathrm{R}(G') \wedge \mathsf{r} \neq \mathsf{r}'\} \overset{s}{\mapsto} \Delta'$, $Q = (\nu s)P$, $s\sharp(\Gamma, c, \Delta)$, $a{:}G \in \Gamma$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash P \triangleright \Delta \cdot \Delta'$. By the induction hypothesis, $\Gamma \vdash P\{d/c\} \triangleright \Delta \cdot \Delta'$. By (ResS), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(MQComR): Then, $Q = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\langle v \rangle^\mathsf{r}\#\mathrm{M}$, $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\langle \mathrm{S} \rangle^\mathsf{r}\#\mathrm{MT}$, $\Gamma \Vdash v{:}\mathrm{S}$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. Because $\Gamma \cdot c{:}\mathrm{S}_c \Vdash v{:}\mathrm{S}$, $s \neq c$. With $\Gamma \Vdash d{:}\mathrm{S}_c$, then $\Gamma \Vdash v\{d/c\}{:}\mathrm{S}$. By the induction hypothesis, $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\{d/xc\} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By (MQComR), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(MQComU): Then, $Q = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l\langle v \rangle^\mathsf{u}\#\mathrm{M}$, $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l'\langle \mathrm{S} \rangle^\mathsf{u}\#\mathrm{MT}$, $l \doteq l'$, $\Gamma \Vdash v{:}\mathrm{S}$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. Because $\Gamma \cdot c{:}\mathrm{S}_c \Vdash v{:}\mathrm{S}$, $s \neq c$. With $\Gamma \Vdash d{:}\mathrm{S}_c$, then $\Gamma \Vdash v\{d/c\}{:}\mathrm{S}$. By the induction hypothesis, $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\{d/xc\} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By (MQComU), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(MQBranR): Then, $Q = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l^\mathsf{r}\#\mathrm{M}$, $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l'^\mathsf{r}\#\mathrm{MT}$, $l \doteq l'$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\{d/xc\} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By (MQBranR), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(MQBranW): Then, $Q = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l^\mathsf{w}\#\mathrm{M}$, $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l'^\mathsf{w}\#\mathrm{MT}$, $l \doteq l'$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\{d/xc\} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By (MQBranW), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(MQDeleg): Then $Q = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}s'[\mathsf{r}]\#\mathrm{M}$, $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}s'[\mathsf{r}]\#\mathrm{MT}$, and $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. Because $\Gamma \cdot c{:}\mathrm{S}_c \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$, $s \neq c$. By the induction hypothesis, $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\{d/xc\} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By (MQDeleg), then $\Gamma \vdash Q\{d/c\} \triangleright \Delta$.

(MQNil): Then, $Q = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}[\,]$ and $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}[\,]$. By (MQNil), then we have $\Gamma \vdash Q\{d/c\} \triangleright \Delta$. $\qquad\square$

Subject reduction tells us that derivatives of well-typed systems are again well-typed. This ensures that our formalism can be used to analyse processes by static type checking. We extend subject reduction such that it provides some information on how the session environment evolves alongside reductions of the system using $\overset{s}{\mapsto}$. In Figure 7 we define the relation $\overset{s}{\mapsto}$ between session environments that emulates the reduction semantics.

Coherence intuitively describes that a session environment captures all local endpoints of a collection of global types. Since we capture all relevant global types in the global environment, we define coherence on pairs of global and session environments.

Definition 5.5 (Coherence)**.** The type environments Γ, Δ are *coherent* if, for all session channels s in Δ, there exists a global type G in Γ such that the restriction of Δ on assignments with s is the set Δ' such that:

$$\{s[\mathsf{r}]{:}G\!\restriction_\mathsf{r} \mid \mathsf{r} \in \mathrm{R}(G)\} \cdot \{s_{\mathsf{r} \to \mathsf{r}'}{:}[\,] \mid \mathsf{r}, \mathsf{r}' \in \mathrm{R}(G)\} \overset{s}{\mapsto} \Delta'$$

We use $\overset{s}{\mapsto}$ in the above definition to define coherence for systems that already performed some steps. We can now prove subject reduction.

Theorem 5.6 (Subject Reduction)**.** *If $\Gamma \vdash P \triangleright \Delta$, Γ, Δ are coherent, and $P \longmapsto P'$, then there is some Δ' such that $\Gamma \vdash P' \triangleright \Delta'$.*

The proof is by induction on the derivation of $P \longmapsto P'$. In every case, we use the information about the structure of the processes to generate partial proof trees for the respective typing judgement. Additionally, we use Condition 5.2.1 to ensure that the type environment of a crashed process cannot contain the types of reliable communication prefixes.

For the proof of subject reduction we further strengthen its goal and show additionally that there is some s such that Γ, Δ' is coherent and $\Delta \overset{s}{\mapsto} \Delta'$, i.e., that the session environment evolves by mimicking the respective reduction step and that this emulation reduces the session environment modulo $\overset{s}{\mapsto}$ w.r.t. a single session s. Moreover we use an additional goal—with weak coherence instead of coherence—to obtain a stronger induction hypothesis for the case of Rule (Par).

Definition 5.7 (Weak Coherence)**.** The type environments Γ, Δ are *weakly coherent* if there exists some Δ' such that $\Gamma, \Delta \cdot \Delta'$ are coherent.

Ultimately, we are however interested into coherence. Note that obviously the coherent case implies the respective weakly coherent case. Our strengthened goal for subject reduction thus becomes:

$$\Gamma \vdash P \triangleright \Delta \wedge \Gamma, \Delta \text{ are coherent} \wedge P \longmapsto P' \longrightarrow$$
$$\exists \Delta'. \, \Gamma \vdash P' \triangleright \Delta' \wedge \Gamma, \Delta' \text{ are coherent} \wedge \Delta \overset{s}{\mapsto} \Delta'$$
$$\text{and}$$
$$\Gamma \vdash P \triangleright \Delta \wedge \Gamma, \Delta \text{ are weakly coherent} \wedge P \longmapsto P' \longrightarrow$$
$$\exists \Delta'. \, \Gamma \vdash P' \triangleright \Delta' \wedge \Gamma, \Delta' \text{ are weakly coherent} \wedge \Delta \overset{s}{\mapsto} \Delta'$$

Proof of Theorem 5.6. The proof is by induction on the reduction $P \longmapsto P'$ that is derived from the rules of the Figures 3 and 4.

Case of Rule (Init): In this case

$$P = \overline{a}[\mathsf{n}](s).Q_{\mathsf{n}} \mid \prod_{1 \leq i \leq n-1} a[i](s).Q_i$$

$$P' = (\nu s)P''$$

$$P'' = \prod_{1 \leq i \leq n} Q_i \mid \prod_{1 \leq i,j \leq n, i \neq j} s_{i \to j}{:}[]$$

$a \neq s$, and we use alpha conversion to ensure that $s \sharp (\Gamma, \Delta)$. By the typing Rules (Par), (Req), and (Acc), $\Gamma \vdash P \triangleright \Delta$ implies that there are G, $\Delta_{Q_1}, \ldots, \Delta_{Q_n}$ such that $\Delta = \Delta_{Q_1} \cdot \ldots \cdot \Delta_{Q_n}$, $a{:}G \in \Gamma$, $\Gamma \vdash Q_i \triangleright \Delta_{Q_i} \cdot s[i]{:}G\restriction_i$ for all $1 \leq i \leq n$, and $|\mathrm{R}(G)| = \mathsf{n}$. By Rule (MQNil), $\Gamma \vdash s_{i \to j}{:}[] \triangleright s_{i \to j}{:}[]$ for all $i, j \in \mathrm{R}(G)$ with $i \neq j$. Since $s \sharp \Delta$, the composition $\Delta \cdot \Delta_s$ for $\Delta_s = \{s[i]{:}G\restriction_i, s_{i \to j}{:}[] \mid i, j \in \mathrm{R}(G) \wedge i \neq j\}$ is defined. By Rule (Par), then $\Gamma \vdash P'' \triangleright \Delta \cdot \Delta_s$. By Rule (Res2), where we use reflexivity to obtain $\Delta_s \overset{s}{\mapsto} \Delta_s$, then $\Gamma \vdash P' \triangleright \Delta$.

Since $\Delta' = \Delta$, Γ, Δ' is coherent and, by reflexivity, $\Delta \overset{s}{\mapsto} \Delta'$.

Case of Rule (RSend): In this case $P = s[\mathsf{r}_1, \mathsf{r}_2]!_\mathsf{r}\langle y\rangle.Q \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}$, $\mathrm{eval}(y) = v$, and $P' = Q \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#\langle v\rangle^\mathsf{r}$. By the Rules (Par), (RSend), and the typing rules for message queues, $\Gamma \vdash P \triangleright \Delta$ implies that there are $\Delta_Q, \mathrm{S}, T, \mathrm{MT}$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_\mathsf{r}\langle \mathrm{S}\rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$, $\Gamma \Vdash y{:}\mathrm{S}$, $\Gamma \vdash Q \triangleright \Delta_Q \cdot s[\mathsf{r}_1]{:}T$, and $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By Rule (MQComR), then $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}\#\langle y\rangle^\mathsf{r} \triangleright s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}\#\langle \mathrm{S}\rangle^\mathsf{r}$. Since $\Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_\mathsf{r}\langle \mathrm{S}\rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$ is defined, so is $\Delta' = \Delta_Q \cdot s[\mathsf{r}_1]{:}T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}\#\langle \mathrm{S}\rangle^\mathsf{r}$. By Rule (Par), then $\Gamma \vdash P' \triangleright \Delta'$.

By Rule (RSend) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (RGet): In this case $P = s[\mathsf{r}_1, \mathsf{r}_2]?_\mathsf{r}(x).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\langle v \rangle^\mathsf{r} \# \mathrm{M}$, $P' = Q\{v/x\} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M}$, and we use alpha conversion to ensure that $x \sharp (\Gamma, \Delta, s)$. By (Par), (RGet), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are $\Delta_Q, \mathrm{S}_1, \mathrm{S}_2, T, \mathrm{MT}$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_\mathsf{r}\langle \mathrm{S}_1 \rangle.T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\langle \mathrm{S}_2 \rangle^\mathsf{r} \# \mathrm{MT}$, $\Gamma \cdot x{:}\mathrm{S}_1 \vdash Q \rhd \Delta_Q \cdot s[\mathsf{r}_1]{:}T$, $\Gamma \Vdash v{:}\mathrm{S}_2$, and $\Gamma \vdash s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M} \rhd s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{MT}$. Since Γ, Δ are coherent, $\mathrm{S}_1 = \mathrm{S}_2$. By Lemma 5.4, then $\Gamma \vdash Q\{v/x\} \rhd \Delta_Q \cdot s[\mathsf{r}_1]{:}T$. Since $\Delta_{Q_j} \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_\mathsf{r}\langle \mathrm{S} \rangle.T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\langle \mathrm{S}_2 \rangle^\mathsf{r} \# \mathrm{MT}$ is defined, so is $\Delta' = \Delta_Q \cdot s[\mathsf{r}_1]{:}T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{MT}$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (RGet) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (RSend): In this case $P = s[\mathsf{r}_1, \mathsf{r}_2]!_\mathsf{u}l\langle y \rangle.Q \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}$, $\mathrm{eval}(y) = v$, and $P' = Q \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \# l\langle v \rangle^\mathsf{u}$. By the Rules (Par), (USend), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are $\Delta_Q, l', \mathrm{S}, T, \mathrm{MT}$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_\mathsf{u}l'\langle \mathrm{S} \rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$, $l \doteq l'$, $l'{:}\mathrm{S} \in \Gamma$, $\Gamma \Vdash y{:}\mathrm{S}$, $\Gamma \vdash Q \rhd \Delta_Q \cdot s[\mathsf{r}_1]{:}T$, and $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \rhd s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By Rule (MQComU), then $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \# l\langle y \rangle^\mathsf{u} \rhd s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT} \# l'\langle \mathrm{S} \rangle^\mathsf{u}$. Since $\Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_\mathsf{u}l'\langle \mathrm{S} \rangle.T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$ is defined, so is $\Delta' = \Delta_Q \cdot s[\mathsf{r}_1]{:}T \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT} \# l'\langle \mathrm{S} \rangle^\mathsf{u}$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (USend) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (UGet): Here $P = s[\mathsf{r}_1, \mathsf{r}_2]?_\mathsf{u}l\langle dv \rangle(x).P \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}l'\langle v \rangle^\mathsf{u} \# \mathrm{M}$, $P' = Q\{v/x\} \mid s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M}$, $l \doteq l'$, and (using alpha conversion) $x \sharp (\Gamma, \Delta, s)$. By (Par), (UGet), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are $\Delta_Q, \mathrm{S}_1, \mathrm{S}_2, T, \mathrm{MT}$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_\mathsf{u}l''\langle \mathrm{S}_1 \rangle.T \cdot s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}l'''\langle \mathrm{S}_2 \rangle^\mathsf{u} \# \mathrm{MT}$, $l \doteq l''$, $l' \doteq l''$, $l''{:}\mathrm{S}_1 \in \Gamma$, $l'''{:}\mathrm{S}_2 \in \Gamma$, $\Gamma \cdot x{:}\mathrm{S}_1 \vdash Q \rhd \Delta_Q \cdot s[\mathsf{r}_1]{:}T$, $\Gamma \Vdash v{:}\mathrm{S}_2$, and $\Gamma \vdash s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{M} \rhd s_{\mathsf{r}_2 \to \mathsf{r}_1}{:}\mathrm{MT}$. Since $l''{:}\mathrm{S}_1 \in \Gamma$, $l'''{:}\mathrm{S}_2 \in \Gamma$, and $l'' \doteq l \doteq l' \doteq l'''$, we have $l'' = l'''$ and $\mathrm{S}_1 = \mathrm{S}_2$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (UGet) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (USkip): In this case $P = s[\mathsf{r}_1, \mathsf{r}_2]?_\mathsf{u}l\langle dv \rangle(x).P$, $P' = Q\{dv/x\}$, and we use alpha conversion to ensure that $x \sharp (\Gamma, \Delta, s)$. By (UGet), $\Gamma \vdash P \rhd \Delta$ implies that there are Δ_Q, S, T such that $\Delta = \Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_\mathsf{u}l'\langle \mathrm{S} \rangle.T$, $l \doteq l'$, $l'{:}\mathrm{S} \in \Gamma$, $\Gamma \Vdash dv{:}\mathrm{S}$, and $\Gamma \cdot x{:}\mathrm{S} \vdash Q \rhd \Delta_Q \cdot s[\mathsf{r}_1]{:}T$. By Lemma 5.4, then $\Gamma \vdash P' \rhd \Delta'$ with $\Delta' = \Delta_Q \cdot s[\mathsf{r}_1]{:}T$.

By Rule (USkip) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (ML): In this case $P = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l\langle v \rangle^\mathsf{u} \# \mathrm{M}$, $P' = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}$. By the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are S, MT such that $\Delta = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}l'\langle \mathrm{S} \rangle^\mathsf{u} \# \mathrm{MT}$, $l \doteq l'$, $l'{:}\mathrm{S} \in \Gamma$, and $\Gamma \vdash P' \rhd \Delta'$ with $\Delta' = s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$.

By Rule (ML) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (RSel): In this case $P = s[\mathsf{r}_1, \mathsf{r}_2]!_\mathsf{r}l.Q \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M}$, and $P' = Q \mid s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \# l^\mathsf{r}$. By the Rules (Par), (RSel), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are $\Delta_Q, \mathrm{I}, j, \mathrm{MT}$ and for all $i \in \mathrm{I}$ there are l_i, T_i such that $\Delta = \Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_\mathsf{r}\{l_i, T_i\}_{i \in \mathrm{I}} \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$, $j \in \mathrm{I}$, $l \doteq l_j$, $\Gamma \vdash Q \rhd \Delta_Q \cdot s[\mathsf{r}_1]{:}T_j$, and $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \rhd s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$. By Rule (MQBranR), then $\Gamma \vdash s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{M} \# l^\mathsf{r} \rhd s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT} \# l_j^\mathsf{r}$. Since $\Delta_Q \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_\mathsf{r}\{l_i, T_i\}_{i \in \mathrm{I}} \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$ is defined, so is $\Delta' = \Delta_Q \cdot s[\mathsf{r}_1]{:}T_j \cdot s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT} \# l_j^\mathsf{r}$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (RSel) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (RBran): In this case $P = s[r_1, r_2]?_r\{l_i.Q_i\}_{i \in I_1} \mid s_{r_2 \to r_1}:l^r\#M$, $j \in I_1$, $l \doteq l_j$, and $P' = Q_j \mid s_{r_2 \to r_1}:M$. By the Rules (Par), (RBran), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are Δ_Q, I_2, MT, l'' and for all $i \in I_2$ there are l'_i, T_i such that $\Delta = \Delta_Q \cdot s[r_1]:[r_2]?_r\{l'_i, T_i\}_{i \in I_2} \cdot s_{r_2 \to r_1}:l''^r\#MT$, $\Gamma \vdash s_{r_2 \to r_1}:M \rhd s_{r_2 \to r_1}:MT$, $l_j \doteq l''$, and for all $k \in I_2$ exists some $m \in I_1$ such that $l_m \doteq l'_k$, $\Gamma \vdash Q_m \rhd \Delta_Q \cdot s[r_1]:T_k$. Since $l \doteq l_j \doteq l''$ and because Γ, Δ are coherent, there is some $n \in I_2$ such that $l_j \doteq l'' = l_n$ and $\Gamma \vdash Q_j \rhd \Delta_Q \cdot s[r_1]:T_n$. Since $\Delta_Q \cdot s[r_1]:[r_2]?_r\{l_i, T_i\}_{i \in I} \cdot s_{r_2 \to r_1}:l''^r\#MT$ is defined, so is $\Delta' = \Delta_Q \cdot s[r_1]:T_n \cdot s_{r_2 \to r_1}:MT$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (RBran) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (WSel): In this case $P = s[r, R]!_w l.Q \mid \prod_{r_i \in R} s_{r \to r_i}:M$ and we have $P' = Q \mid \prod_{r_i \in R} s_{r \to r_i}:M_i \# l^w$. Let $R = \{r_1, \ldots, n\}$. By the Rules (Par), (WSel), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are $\Delta_Q, I, j, MT_1, \ldots, MT_n$ and for all $i \in I$ there are l_i, T_i such that $\Delta = \Delta_Q \cdot s[r]:[R]!_w\{l_i, T_i\}_{i \in I} \cdot s_{r \to r_1}:MT \cdot \ldots \cdot s_{r \to r_n}:MT$, $j \in I$, $l \doteq l_j$, $\Gamma \vdash Q \rhd \Delta_Q \cdot s[r]:T_j$, and $\Gamma \vdash s_{r \to r_1}:M_1 \rhd s_{r \to r_1}:MT_1$, $\ldots$, $\Gamma \vdash s_{r \to r_n}:M_n \rhd s_{r \to r_n}:MT_n$. By Rule (MQBranW), then $\Gamma \vdash s_{r \to r_1}:M_1\#l^w \rhd s_{r \to r_1}:MT_1\#l_j^w$, $\ldots$, $\Gamma \vdash s_{r \to r_n}:M_n\#l^w \rhd s_{r \to r_n}:MT_n\#l_j^w$. Since $\Delta_Q \cdot s[r]:[R]!_w\{l_i, T_i\}_{i \in I} \cdot s_{r \to r_1}:MT_1 \cdot \ldots \cdot s_{r \to r_n}:MT_n$ is defined, so is $\Delta' = \Delta_Q \cdot s[r]:T_j \cdot s_{r \to r_1}:MT_1\#l_j^w \cdot \ldots \cdot s_{r \to r_n}:MT_n\#l_j^w$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (WSel) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (WBran): Here $P = s[r_1, r_2]?_w\{l_i.Q_i\}_{i \in I_1, l_d} \mid s_{r_2 \to r_1}:l^w\#M$, $j \in I_1$, $l \doteq l_j$, and $P' = Q_j \mid s_{r_2 \to r_1}:M$. By the Rules (Par), (WBran), and the typing rules for message queues, $\Gamma \vdash P \rhd \Delta$ implies that there are $\Delta_Q, l'_d, I_2, MT, l''$ and for all $i \in I_2$ there are l'_i, T_i such that $\Delta = \Delta_Q \cdot s[r_1]:[r_2]?_w\{l'_i, T_i\}_{i \in I_2, l'_d} \cdot s_{r_2 \to r_1}:l''^w\#MT$, $l_d \doteq l'_d$, $\Gamma \vdash s_{r_2 \to r_1}:M \rhd s_{r_2 \to r_1}:MT$, $l_j \doteq l''$, and for all $k \in I_2$ exists some $m \in I_1$ such that $l_m \doteq l'_k$, $\Gamma \vdash Q_m \rhd \Delta_Q \cdot s[r_1]:T_k$. Since $l \doteq l_j \doteq l''$ and because Γ, Δ are coherent, there is some $n \in I_2$ such that $l_j \doteq l'' = l_n$ and $\Gamma \vdash Q_j \rhd \Delta_Q \cdot s[r_1]:T_n$. Since $\Delta_Q \cdot s[r_1]:[r_2]?_w\{l_i, T_i\}_{i \in I, l'_d} \cdot s_{r_2 \to r_1}:l''^w\#MT$ is defined, so is $\Delta' = \Delta_Q \cdot s[r_1]:T_n \cdot s_{r_2 \to r_1}:MT$. By Rule (Par), then $\Gamma \vdash P' \rhd \Delta'$.

By Rule (WBran) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (WSkip): In this case $P = s[r_1, r_2]?_w\{l_i.Q_i\}_{i \in I_1, l_d}$ and $P' = Q_d$. By the Rule(WBran), $\Gamma \vdash P \rhd \Delta$ implies that there are Δ_Q, l'_d, I_2 and for all $i \in I_2$ there are l'_i, T_i such that $\Delta = \Delta_Q \cdot s[r_1]:[r_2]?_w\{l'_i, T_i\}_{i \in I_2, l'_d}$, $l_d \doteq l'_d$, and for all $k \in I_2$ exists some $m \in I_1$ such that $l_m \doteq l'_k$, $\Gamma \vdash Q_m \rhd \Delta_Q \cdot s[r_1]:T_k$. Since $l_d \doteq l'_d$, there is some $n \in I_2$ such that $l_d \doteq l'_d = l_n$ and $\Gamma \vdash P' \rhd \Delta'$ with $\Delta' = \Delta_Q \cdot s[r_1]:T_d$.

By Rule (WSkip) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (Crash): In this case $\texttt{FP}_{\texttt{crash}}$ and $P' = \bot$. By the typing rules and Condition 5.2.1, $\Gamma \vdash P \rhd \Delta$ and $\texttt{FP}_{\texttt{crash}}$ imply $\text{nsr}(\Delta)$. By Rule (Crash), then $\Gamma \vdash \bot \rhd \Delta$. Since $\Delta' = \Delta$, Γ, Δ' is coherent and, by reflexivity, $\Delta \overset{s}{\mapsto} \Delta'$.

Case of Rule (If-T): In this case $P = \texttt{if } e \texttt{ then } Q \texttt{ else } Q'$ and $P' = Q$. By Rule (If), $\Gamma \vdash P \triangleright \Delta$ implies $\Gamma \vdash P' \triangleright \Delta$.

Since $\Delta' = \Delta$, Γ, Δ' is coherent and, by reflexivity, $\Delta \overset{s}{\Rightarrow} \Delta'$.

Case of Rule (If-F): In this case $P = \texttt{if } e \texttt{ then } Q \texttt{ else } Q'$ and $P' = Q'$. By Rule (If), $\Gamma \vdash P \triangleright \Delta$ implies $\Gamma \vdash P' \triangleright \Delta$.

Since $\Delta' = \Delta$, Γ, Δ' is coherent and, by reflexivity, $\Delta \overset{s}{\Rightarrow} \Delta'$.

Case of Rule (Deleg): In this case $P = s[\mathsf{r_1}, \mathsf{r_2}]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.Q \mid s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{M}$ and $P' = Q \mid s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{M}\#s'[\mathsf{r}]$. By the Rules (Par), (Deleg), and the typing rules for message queues, $\Gamma \vdash P \triangleright \Delta$ implies that there are $\Delta_Q, T, T', \mathrm{MT}$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r_1}]{:}[\mathsf{r2}]!\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s'[\mathsf{r}]{:}T' \cdot s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{MT}$, $\Gamma \vdash Q \triangleright \Delta_Q \cdot s[\mathsf{r_1}]{:}T$, and $\Gamma \vdash s_{\mathsf{r_1}\to\mathsf{r_s}}{:}\mathrm{MT} \triangleright s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{MT}$. By Rule (MQDeleg), then $\Gamma \vdash s_{\mathsf{r_1}\to\mathsf{r_s}}{:}\mathrm{MT}\#s'[\mathsf{r}] \triangleright s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{MT}\#s'[\mathsf{r}]$. Since $\Delta = \Delta_Q \cdot s[\mathsf{r_1}]{:}[\mathsf{r2}]!\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s'[\mathsf{r}]{:}T' \cdot s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{MT}$ is defined, so is $\Delta' = \Delta_Q \cdot s[\mathsf{r_1}]{:}T \cdot s_{\mathsf{r_1}\to\mathsf{r_2}}{:}\mathrm{MT}\#s'[\mathsf{r}]$. By Rule (Par), then $\Gamma \vdash P' \triangleright \Delta'$.

By Rule (Deleg) of Figure 7, then $\Delta \overset{s}{\Rightarrow} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\Rightarrow}$, then Γ, Δ' are coherent.

Case of Rule (SRecv): In this case $P = s[\mathsf{r_1}, \mathsf{r_2}]?(\!(s'[\mathsf{r}])\!).Q \mid s_{\mathsf{r_2}\to\mathsf{r_1}}{:}s''[\mathsf{r'}]\#\mathrm{M}$ and $P' = Q\{s''/s'\}\{\mathsf{r'}/\mathsf{r}\} \mid \mathrm{M}$. We use alpha conversion to ensure that $s' = s''$ and $\mathsf{r} = \mathsf{r''}$. By the Rules (Par), (SRecv), and the typing rules for message queues, $\Gamma \vdash P \triangleright \Delta$ implies that there are $\Delta_Q, T, T', \mathrm{MT}$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r_1}]{:}[\mathsf{r_2}]?\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s_{\mathsf{r_2}\to\mathsf{r_1}}{:}s'[\mathsf{r}]\#\mathrm{MT}$, $\Gamma \vdash Q \triangleright \Delta_Q \cdot s[\mathsf{r_1}]{:}T \cdot s'[\mathsf{r}]{:}T'$, and $\Gamma \vdash s_{\mathsf{r_2}\to\mathsf{r_1}}{:}\mathrm{M} \triangleright s_{\mathsf{r_2}\to\mathsf{r_1}}{:}\mathrm{MT}$. Since $\Delta = \Delta_Q \cdot s[\mathsf{r_1}]{:}[\mathsf{r_2}]?\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s_{\mathsf{r_2}\to\mathsf{r_1}}{:}s''[\mathsf{r'}]\#\mathrm{MT}$ is defined, so is $\Delta' = \Delta_Q \cdot s[\mathsf{r_1}]{:}T \cdot s'[\mathsf{r}]{:}T' \cdot s_{\mathsf{r_2}\to\mathsf{r_1}}{:}\mathrm{MT}$. By Rule (Par), then $\Gamma \vdash P' \triangleright \Delta'$.

By Rule (SRecv) of Figure 7, then $\Delta \overset{s}{\Rightarrow} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\Rightarrow}$, then Γ, Δ' are coherent.

Case of Rule (Par): In this case $P = Q_1 \mid Q_2$, $Q_1 \longmapsto Q_1'$, and $P' = Q_1' \mid Q_2$. By Rule (Par), $\Gamma \vdash P \triangleright \Delta$ implies that there are $\Delta_{Q_1}, \Delta_{Q_2}$ such that $\Delta = \Delta_{Q_1} \cdot \Delta_{Q_2}$, $\Gamma \vdash Q_1 \triangleright \Delta_{Q_1}$, and $\Gamma \vdash Q_2 \triangleright \Delta_{Q_2}$. Since Γ, Δ are coherent, Γ, Δ_{Q_1} is weakly coherent. By the induction hypothesis (for the weakly coherent case), $\Gamma \vdash Q_1' \triangleright \Delta'_{Q_1}$ with $\Delta_{Q_1} \overset{s}{\Rightarrow} \Delta'_{Q_1}$. Since $\Delta_{Q_1} \cdot \Delta_{Q_2}$ is defined, so is $\Delta' = \Delta_{Q_1} \cdot \Delta_{Q_2}$. By Rule (Par), then $\Gamma \vdash P' \triangleright \Delta'$.

By $\Delta_{Q_1} \overset{s}{\Rightarrow} \Delta'_{Q_1}$, then $\Delta \overset{s}{\Rightarrow} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\Rightarrow}$, then Γ, Δ' are coherent.

Case of Rule (Res): In this case $P = (\nu x)Q$, $Q \longmapsto Q'$, and $P' = (\nu x)Q'$. Then, one of the Rules (Res1) or (Res2) was used to type the restriction on x in P.

Case of (Res1): Then there is some S such that $x\sharp(\Gamma, \Delta)$ and $\Gamma \cdot x{:}\mathrm{S} \vdash Q \triangleright \Delta$. Since Γ, Δ are coherent, so are $\Gamma \cdot x{:}\mathrm{S}, \Delta$. By the induction hypothesis, $\Gamma \cdot x{:}\mathrm{S} \vdash Q' \triangleright \Delta'$ for some Δ' such that Γ, Δ' is coherent and $\Delta \overset{s}{\Rightarrow} \Delta'$. With Rule (Res1), then $\Gamma \vdash P' \triangleright \Delta'$.

Case of (Res2): In this case $x = s$ and there are G, a, Δ'' such that we have $s\sharp(\Gamma, \Delta)$, $\{s[\mathsf{r}]{:}G{\restriction}_\mathsf{r} \mid \mathsf{r} \in \mathrm{R}(G)\} \cdot \{s_{\mathsf{r}\to\mathsf{r'}}{:}[] \mid \mathsf{r}, \mathsf{r'} \in \mathrm{R}(G') \wedge \mathsf{r} \neq \mathsf{r'}\} \overset{s}{\Rightarrow} \Delta''$, $a{:}G \in \Gamma$, and $\Gamma \vdash Q \triangleright \Delta \cdot \Delta''$. Then $\Gamma, \Delta \cdot \Delta''$ are coherent. By the induction hypothesis, $\Gamma \vdash Q' \triangleright \Delta'''$ for some Δ''' such that $\Delta \cdot \Delta'' \overset{s}{\Rightarrow} \Delta'''$. By Rule (Res2), then $\Gamma \vdash P' \triangleright \Delta$. Since $\Delta' = \Delta$, Γ, Δ' is coherent and, by reflexivity, $\Delta \overset{s}{\Rightarrow} \Delta'$.

Case of (Rec): In this case $P = (\mu X)Q$ and $P' = Q\{(\mu X)Q/X\}$. By Rule (Rec), then there are $\Delta_Q, s, \mathsf{r}, t, T$ such that $\Delta = \Delta_Q \cdot s[\mathsf{r}]{:}(\mu t)T$ and $\Gamma \cdot X{:}s[\mathsf{r}]{:}t \vdash Q \triangleright \Delta_Q \cdot s[\mathsf{r}]{:}T$. By

replacing in the proof tree of $\Gamma \cdot X{:}s[\mathsf{r}]{:}t \vdash Q \triangleright \Delta_Q \cdot s[\mathsf{r}]{:}T$ all occurrences of Rule (Var) by the proof tree, we obtain $\Gamma \vdash P' \triangleright \Delta'$ with $\Delta' = \Delta_Q \cdot s[\mathsf{r}]{:}T\{^{(\mu t)T}/t\}$.

By Rule (Rec) of Figure 7, then $\Delta \overset{s}{\mapsto} \Delta'$. Since Γ, Δ are coherent and by transitivity of $\overset{s}{\mapsto}$, then Γ, Δ' are coherent.

Case of Rule (Struc): In this case $P \equiv Q$, $Q \longmapsto Q'$, $Q' \equiv P'$. By Lemma 5.3, $\Gamma \vdash P \triangleright \Delta$ and $P \equiv Q$ imply $\Gamma \vdash Q \triangleright \Delta$. By the induction hypothesis, $\Gamma \vdash Q' \triangleright \Delta'$ for some Δ' such that $\Delta \overset{s}{\mapsto} \Delta'$ and Γ, Δ' are coherent. By Lemma 5.3, then $\Gamma \vdash P' \triangleright \Delta'$. □

Since we restrict our attention to linear environments, type judgements ensure linearity of session channels. With subject reduction, this holds for all derivatives of well-typed processes.

Lemma 5.8 (Linearity). *Let $\Gamma \vdash P \triangleright \Delta$, Γ, Δ be coherent, and there are no name clashes on session channels. Then all session channels of P are linear, i.e., for all $P \longmapsto^* P'$ and all $s, \mathsf{r}, \mathsf{r}_1, \mathsf{r}_2$ there is at most one unguarded actor $s[\mathsf{r}]$ and at most one queue $s_{\mathsf{r}_1 \to \mathsf{r}_2}$ in P'.*

Proof. By Theorem 5.6, there is some Δ' such that $\Gamma \vdash P' \triangleright \Delta'$ and Γ, Δ' are coherent. By the Definition 5.5 of coherence and projection, Δ' contains at most one actor $s[\mathsf{r}]$ and at most one queue $s_{\mathsf{r}_1 \to \mathsf{r}_2}$ for each $a{:}G \in \Gamma$ and $\mathsf{r} \in \mathsf{R}(G)$. By the Figures 5 and 6, only the Rules (Req), (Acc), and (Res2) can introduce new actors or queues. The linearity of global environments ensures, that all new actors and queues introduced by the rules are on fresh channel names and are pairwise distinct. The Rules (Req) and (Acc) introduce exactly one actor each on a fresh session channel s that is bound by a prefix for session initialisation. Rule (Res2) introduces assignments for actors and queues for pairwise different roles on a fresh session channel s that is bound by restriction. Since there are no name clashes, the session channels in binders are pairwise different and distinct from free session channels. By the typing rules and because $\Gamma \vdash P' \triangleright \Delta'$, all actors and queues in P' have to satisfy their specification as described by an assignment of this actor or queue towards a local type. By the linearity of session environments and since new assignments for actors result from bound session channels, all unguarded actors and queues in P' are pairwise different. □

For strongly reliable systems coherence ensures that for each actor there is a matching communication partner. In the case of asynchronous communication, this means that for each sender (or message on a queue) there is a receiver and for each receiver there is a sender or a message on a queue, where the receiver as well as the sender or the message queue appear under the same binder of the session channel or both are free. In the case of unreliable communication, messages get lost, senders can crash, and receivers can crash themselves or suspect the sender. In the case of weakly reliable branching for each sender (or message on a queue) there are all specified receivers that are not crashed and vice versa.

We summarise these properties of strongly reliable and weakly reliable interactions in *error-freedom*: for each strongly reliable sender or message there is a matching receiver and vice versa, for each weakly reliable sender or message there is a possibly crashed receiver and vice versa. We obtain similar requirements for session delegation.

Lemma 5.9 (Error-Freedom). *If $\Gamma \vdash P \triangleright \Delta$ and Γ, Δ is coherent then:*
- *for each unguarded $s[\mathsf{r}_1, \mathsf{r}_2]!_\mathsf{r}\langle y \rangle.Q_1$ and each message $\langle y \rangle^\mathsf{r}$ on a message queue $s_{\mathsf{r}_1 \to \mathsf{r}_2}$ in P there is some $s[\mathsf{r}_2, \mathsf{r}_1]?_\mathsf{r}(x).Q_2$ in P,*
- *for each unguarded $s[\mathsf{r}_2, \mathsf{r}_1]?_\mathsf{r}(x).Q_2$ in P there is some $s[\mathsf{r}_1, \mathsf{r}_2]!_\mathsf{r}\langle y \rangle.Q_1$ or a message $\langle y \rangle^\mathsf{r}$ on a message queue $s_{\mathsf{r}_1 \to \mathsf{r}_2}$ in P,*

- *for each unguarded $s[r_1, r_2]!_r l.Q$ and each message l^r on a message queue $s_{r_1 \to r_2}$ in P there is some $j \in I$ and $s[r_2, r_1]?_r\{l_i.Q_i\}_{i \in I}$ in P with $l_j \doteq l$,*
- *for each unguarded $s[r_2, r_1]?_r\{l_i.Q_i\}_{i \in I}$ in P there is $j \in I$ and $s[r_1, r_2]!_r l.Q$ or a message l^r on a message queue $s_{r_1 \to r_2}$ in P with $l_j \doteq l$,*
- *for each unguarded $s[r, R]!_w l.Q$ and each message l^w on a message queue $s_{r \to r'}$ in P and each $r' \in R$ there is some $s[r', r]?_w\{l_i.P_i\}_{i \in I, l_d}$ and $j \in I$ in P with $l_j \doteq l$ or P does not contain an actor $s[r']$,*
- *for each unguarded $s[r', r]?_w\{l_i.P_i\}_{i \in I, l_d}$ in P there is $j \in I$ and $s[r, R]!_w l.Q$ or a message l^w on a message queue $s_{r \to r'}$ in P with $l_j \doteq l$ and $r' \in R$ or P does not contain an actor $s[r]$,*
- *for each unguarded $s[r_1, r_2]!\langle\!\langle s'[r]\rangle\!\rangle.Q_1$ and each message $s'[r]$ on a message queue $s_{r_1 \to r_2}$ in P there is some $s[r_2, r_1]?(\!(s''[r'])\!).Q_2$ in P, and*
- *for each unguarded $s[r_2, r_1]?(\!(s''[r'])\!).Q_2$ in P there is some $s[r_1, r_2]!\langle\!\langle s'[r]\rangle\!\rangle.Q_1$ or a message $s'[r]$ on a message queue $s_{r_1 \to r_2}$ in P.*

Proof. By coherence and projection for each strongly reliable and each weakly reliable sender there is initially a matching receiver for each free session channel in the session environment. By the typing rules and Rule (Res2) in particular, this holds also for restricted session channels. Session environments may evolve using $\overset{s}{\mapsto}$ but all such steps preserve the above defined requirements, i.e., strongly reliable or weakly reliable send prefixes can be mapped onto the type of the respective message in a queue but no such message can be dropped. The typing rules ensure that the processes follow their specification in the local types of session environments. Then, the first four and the last two cases follow from the typing rules and coherence, and the fact that only unreliable processes can crash. The remaining two cases follow from the typing rules and coherence. □

Session fidelity claims that the interactions of a well-typed process follow exactly the specification described by its global types, i.e., if a system is well-typed w.r.t. to coherent type environments then the system follows its specification in the global type. One direction of this property already follows from the above variant of subject reduction. The steps of well-typed systems are reflected by corresponding steps of the session environment and, thus, respect their specification in global types. What remains to show is that the specified interactions can indeed be performed. The above formulation of error-freedom alone is not strong enough to show this, because it ensures only the existence of matching communication partners and not that they can be unguarded.

To obtain session fidelity we prove *progress*. *Progress* states that no part of a well-typed and coherent system can block other parts, that eventually all matching communication partners as described by error-freedom are unguarded, that interactions specified by the global type can happen, and that there are no communication mismatches. Subject reduction and progress together then imply *session fidelity*, i.e., that processes behave as specified in their global types.

To ensure that the interleaving of sessions and session delegation cannot introduce deadlocks, we assume an interaction type system as introduced in [BCD+08, HYC16]. For this type system it does not matter whether the considered actions are strongly reliable, weakly reliable, or unreliable. More precisely, we can adapt the interaction type system of [BCD+08] in a straightforward way to the above session calculus, where unreliable communication and weakly reliable branching is treated in exactly the same way as strongly reliable communication/branching. We say that P *is free of cyclic dependencies between*

sessions if this interaction type system does not detect any cyclic dependencies. In this sense fault-tolerance is more flexible than explicit failure handling, which oftens requires a more substantial revision of the interaction type system to cover the additional dependencies that are introduced e.g. by the propagation of faults.

In the literature there are different formulations of progress. We are interested in a rather strict definition of progress that ensures that well-typed systems cannot block. Therefore, we need an additional assumption on session requests and acceptances. Coherence ensures the existence of communication partners within sessions only. If we want to avoid blocking, we need to be sure, that no participant of a session is missing during its initialisation. Note that without action prefixes all participants either terminated or crashed.

Theorem 5.10 (Progress/Session Fidelity). *Let $\Gamma \vdash P \triangleright \Delta$, Γ, Δ be coherent, and let P be free of cyclic dependencies between sessions. Assume that in the derivation of $\Gamma \vdash P \triangleright \Delta$, whenever $\overline{a}[\mathsf{n}](s).Q$ or $a[\mathsf{r}](s).Q$ in P, then $a{:}G \in \Gamma$, $|\mathrm{R}(G)| = \mathsf{n}$, and there are $\overline{a}[\mathsf{n}](s).Q_n$ as well as $a[\mathsf{r}_i](s).Q_i$ in P for all $1 \leq \mathsf{r}_i < \mathsf{n}$.*
(1) Then either P does not contain any action prefixes or $P \longmapsto P'$.
(2) If P does not contain recursion, then there exists P' such that $P \longmapsto^* P'$ and P' does not contain any action prefixes.

Proof. 1.) If P contains an unguarded conditional, then it can perform a step $P \longmapsto P'$ such that $\Gamma \vdash P' \triangleright \Delta$ using one of the Rules (If-T) or (If-F) as described in the corresponding cases of the proof of Theorem 5.6. Similarly, if P contains an unguarded recursion, then it can perform a step $P \longmapsto P'$ such that $\Gamma \vdash P' \triangleright \Delta'$ with $\Delta \overset{s}{\mapsto} \Delta'$ using Rule (Rec) as described in the corresponding case of the proof of Theorem 5.6. Assume that P is not structural congruent to **0** and does not contain unguarded conditionals or recursions.

Since P is not structural congruent to **0** it contains session channels. All session channels of P that are not contained in Δ are bound in P, i.e., the names of Δ are exactly the free session channels of P. Since there are no cyclic dependencies between sessions, we can pick a minimal session in P, i.e., a session such that its next action is not blocked by any action of another session or session delegation. Let s denote this session channel. By the typing rules in the Figures 5 and 6, there are some G, b such that $b{:}G \in \Gamma$ and G specifies the session s. Since the next action of this session is not blocked, P contains at least one unguarded prefix or at least one unguarded not empty queue on s. Among all such unguarded prefixes and messages that are head of a queue we pick a minimal, i.e., one that is typed by the projection of a part of G such that no part of G that is guarding it is used to type another unguarded prefix or message in P:

$\overline{a}[\mathsf{n}](s).Q$**:** Then $s \sharp \Delta$ and $a{:}G \in \Gamma$. By the assumption on session initialisation, then for all $1 \leq \mathsf{r}_i < \mathsf{n}$ we have $a[\mathsf{r}_i](s).$ in P. Since s is minimal, all these session acceptances are unguarded. By Rule (Init), then $P \longmapsto P'$.

$a[\mathsf{r}](s).Q$**:** Then $s \sharp \Delta$ and $a{:}G \in \Gamma$. By the assumption on session initialisation, then $\overline{a}[\mathsf{n}](s).Q$ and for all $1 \leq \mathsf{r}_i < \mathsf{n}$ with $\mathsf{r}_i \neq \mathsf{r}$ we have $a[\mathsf{r}_i](s).$ in P. Since s is minimal, all this session request and these session acceptances are unguarded. By Rule (Init), then $P \longmapsto P'$.

$s[\mathsf{r}_1, \mathsf{r}_2]!_\mathsf{r}\langle y \rangle.Q$**:** If $s \sharp \Delta$ then the typing rules in the Figures 5 and 6 and (Req), (Acc), and (Res2) in particular ensure that $s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$ in P. If s is free in Δ, then $s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$ in P because of coherence. Since s and the action are minimal, $s_{\mathsf{r}_1 \to \mathsf{r}_2}{:}\mathrm{MT}$ is unguarded. By Rule (RSend), then $P \longmapsto P'$.

$s_{r_1 \to r_2} : \langle v \rangle^r \# M$: By Lemma 5.9, then $s[r_2, r_1]?_r(x).Q$ in P. Since s and the action are minimal, $s[r_2, r_1]?_r(x).Q$ is unguarded. By Rule (RGet), then $P \longmapsto P'$.

$s[r_2, r_1]?_r(x).Q$: By Lemma 5.9, then $s_{r_1 \to r_2} : \langle v \rangle^r \# M$ in P. Since s and the action are minimal, $s_{r_1 \to r_2} : \langle v \rangle^r \# M$ is unguarded. By Rule (RGet), then $P \longmapsto P'$.

$s[r_1, r_2]!_u l\langle y \rangle.Q$: If $s \sharp \Delta$ then the typing rules in the Figures 5 and 6 and (Req), (Acc), and (Res2) in particular ensure that $s_{r_1 \to r_2}:MT$ in P. If s is free in Δ, then $s_{r_1 \to r_2}:MT$ in P, because of coherence. Since s and the action are minimal, $s_{r_1 \to r_2}:MT$ is unguarded. By Rule (USend), then $P \longmapsto P'$.

$s_{r_1 \to r_2} : l\langle v \rangle^u \# M$: Then the typing rules and coherence ensure that either there is no actor $s[r_2]$, the actor $s[r_2]$ skipped, or $s[r_2, r_1]?_u l\langle dv \rangle(x).Q$ in P. If there is no actor $s[r_2]$, then it has crashed. Then $FP_{crash}(Q, \ldots)$ was satisfied with $s[r_2] \in A(Q)$. By Condition 5.2.5, then eventually $FP_{ml}(s, r_1, r_2, l, \ldots)$. By Rule (ML), then $P \longmapsto P'$. If the actor $s[r_2]$ skipped this reception, then $FP_{uskip}(s, r_2, r_1, l, \ldots)$. By Condition 5.2.3, then $FP_{ml}(s, r_1, r_2, l, \ldots)$. By Rule (ML), then $P \longmapsto P'$. If $s[r_2, r_1]?_u l\langle dv \rangle(x).Q$ in P, since s and the action are minimal, then this term is unguarded. By Condition 5.2.2 and Rule (UGet), then $P \longmapsto P'$.

$s[r_2, r_1]?_u l\langle dv \rangle(x).Q$: Then the typing rules and coherence ensure that there is the queue $s_{r_1 \to r_2}$ and either $l\langle v \rangle^u$ is on top of it, or this message was dropped, or the sender did not yet send this message, or the sender crashed before transmitting this message. If $l\langle v \rangle^u$ is on top of the queue, then $P \longmapsto P'$ by Rule (UGet) and Condition 5.2.2. If the message was dropped, then $FP_{ml}(s, r_1, r_2, l, \ldots)$. By Condition 5.2.3, then $FP_{uskip}(s, r_2, r_1, l, \ldots)$. By Rule (USkip), then $P \longmapsto P'$. If the sender did not yet send the message, we proceed as in the Case $s[r_1, r_2]!_u l\langle y \rangle.Q$ above. If the sender crashed, then $FP_{crash}(Q, \ldots)$ with $s[r_1] \in A(Q)$. By Condition 5.2.4, then eventually $FP_{uskip}(s, r_2, r_1, l, \ldots)$. By Rule (USkip), then $P \longmapsto P'$.

$s[r_1, r_2]!_r l.Q$: If $s \sharp \Delta$ then the typing rules in the Figures 5 and 6 and (Req), (Acc), and (Res2) in particular ensure that $s_{r_1 \to r_2}:MT$ in P. If s is free in Δ, then $s_{r_1 \to r_2}:MT$ in P, because of coherence. Since s and the action are minimal, $s_{r_1 \to r_2}:MT$ is unguarded. By Rule (RSel), then $P \longmapsto P'$.

$s_{r_1 \to r_2} : l^r \# M$: By Lemma 5.9, then $s[r_2, r_1]?_r\{l_i.Q_i\}_{i \in I}$ in P. Since s and the action are minimal, $s[r_2, r_1]?_r\{l_i.Q_i\}_{i \in I}$ is unguarded. By (RBran), then $P \longmapsto P'$.

$s[r_2, r_1]?_r\{l_i.Q_i\}_{i \in I}$: By Lemma 5.9, then $s_{r_1 \to r_2} : l^r \# M$ in P with $j \in I$ and $l \doteq l_j$. Since s and the action are minimal, $s_{r_1 \to r_2} : l^r \# M$ is unguarded. By (RBran), then $P \longmapsto P'$.

$s[r, R]!_w l.Q$: If $s \sharp \Delta$ then the typing rules and (Req), (Acc), and (Res2) in particular ensure that $s_{r \to r_i}:MT_i$ in P for all $r_i \in R$. If s is free in Δ, then $s_{r \to r_i}:MT_i$ in P for all $r_i \in R$, because of coherence. Since s and the action are minimal, all $s_{r \to r_i}:MT_i$ are unguarded. By Rule (WSel), then $P \longmapsto P'$.

$s_{r_1 \to r_2} : l^w \# M$: Then the typing rules, coherence, and Condition 5.2.6 ensure that either there is no actor $s[r_2]$ or $s[r_2, r_1]?_w\{l_i.Q_i\}_{i \in I, l_d}$ in P. In the former case the actor $s[r_2]$ has crashed. Then, since the message queue will not be needed any more, proceed with the next session and action that are minimal if you ignore the queue $s_{r_1 \to r_2}$.

$s[r_2, r_1]?_w\{l_i.Q_i\}_{i \in I, l_d}$: Then the typing rules and coherence ensure that there is the queue $s_{r_1 \to r_2}$ and either l^w is on top of it, or the sender did not yet send this message, or the sender crashed before transmitting this message. If l^w is on top of the queue, then $P \longmapsto P'$ by Rule (WBran). If the sender did not yet send the message, we proceed as in the Case $s[r, R]!_w l.Q$ above. If the sender crashed, then $FP_{crash}(Q', \ldots)$ with

$s[\mathsf{r}_1] \in Q'$. By Condition 5.2.4, then eventually $\mathsf{FP}_{\mathtt{wskip}}(s, \mathsf{r}_2, \mathsf{r}_1, l, \ldots)$ for $s[\mathsf{r}_2]$. By Rule (WSkip), then $P \longmapsto P'$.

$s[\mathsf{r}_1, \mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.Q$**:** If $s\sharp\Delta$ then the typing rules in the Figures 5 and 6 and (Req), (Acc), and (Res2) in particular ensure that $s_{\mathsf{r}_1\to\mathsf{r}_2}$:MT in P. If s is free in Δ, then $s_{\mathsf{r}_1\to\mathsf{r}_2}$:MT in P because of coherence. Since s and the action are minimal, $s_{\mathsf{r}_1\to\mathsf{r}_2}$:MT is unguarded. By Rule (Deleg), then $P \longmapsto P'$.

$s_{\mathsf{r}_1\to\mathsf{r}_2}$:$s'[\mathsf{r}]\#$M**:** By Lemma 5.9, then $s[\mathsf{r}_2, \mathsf{r}_1]?(\!(s''[\mathsf{r}'])\!).Q$ in P. Since s and the action are minimal, $s[\mathsf{r}_2, \mathsf{r}_1]?(\!(s''[\mathsf{r}'])\!).Q$ is unguarded. By Rule (SRecv), then $P \longmapsto P'$.

$s[\mathsf{r}_2, \mathsf{r}_1]?(\!(s''[\mathsf{r}'])\!).Q$**:** By Lemma 5.9, then $s_{\mathsf{r}_1\to\mathsf{r}_2}$:$s'[\mathsf{r}]\#$M in P. Since s and the action are minimal, $s_{\mathsf{r}_1\to\mathsf{r}_2}$:$s'[\mathsf{r}]\#$M is unguarded. By Rule (SRecv), then $P \longmapsto P'$.

2.) By Theorem 5.6, P' is well-typed w.r.t. coherent Γ, Δ' with $\Delta \overset{s}{\Mapsto} \Delta'$. If P does not contain recursion, then the session environment strictly reduces with every reduction of the process that does not reduce a conditional, though it may grow in cases of session initialisation. Since P is finite and loop-free there are only finitely many possible session initialisations. We can repeat the above proof for 1.) to show that P cannot get stuck as long as it contains action prefixes. □

The proof of progress relies on the Condition 5.2.2–5.2.6 to ensure that failures cannot block the system: in the failure-free case unreliable messages are eventually received (5.2.2), the receiver of a lost message can skip (5.2.3), no receiver is blocked by a crashed sender (5.2.4), messages towards receivers that crashed or skipped can be dropped (5.2.5 + 5.2.3), and branching requests cannot be ignored (5.2.6).

6. The Rotating Coordinator Algorithm

To illustrate our approach we study a Consensus algorithm by Chandra and Toueg (cf. [CT96, FMN07]). This algorithm is small but not trivial. It was designed for systems with crash failures, but the majority of the algorithm can be implemented with unreliable communication.

As this algorithm models consensus, the goal is that every agent i eventually decides on a proposed belief value, where no two agents decide on different values. It is a round based algorithm, where each round consists of four phases. In each round, one process acts as a coordinator decided by round robin, denoted by c.

In Phase 1: every agent i sends its current belief to the coordinator c.

In Phase 2: the coordinator waits until it has received at least half of the messages of the current round and then sends the best belief to all other agents.

In Phase 3: the agents either receive the message of the coordinator or suspect the coordinator to have crashed and reply with ack or nack accordingly. Suspicion can yield false positives.

In Phase 4: the coordinator waits, as in Phase 2, until it has received at least half of the messages of the current round and then sends a weakly reliable broadcast if at least half of the messages contained ack.

It is possible for agents to skip rounds by suspecting the coordinator of the current round and by proceeding to the next round. There are also no synchronisation fences thus it is possible for the agents to be in different rounds and have messages of different rounds in the system. Having agents in different rounds makes proving correctness much more difficult.

6.1. **Specification and Implementation.** Let $\left(\bigodot_{1\leq i\leq n}\pi_i\right).G$ abbreviate $\pi_1.\ldots.\pi_n.G$ to simplify the presentation, where $G \in \mathcal{G}$ is a global type and $\pi_1, \ldots, \pi_n$ are sequences of prefixes. More precisely, each π_i is of the form $\pi_{i,1}.\ldots.\pi_{i,m}$ and each $\pi_{i,j}$ is a type prefix of the form $\mathsf{r}_1 \to_\mathsf{u} \mathsf{r}_2{:}l\langle\mathsf{S}\rangle$ or $\mathsf{r} \to_\mathsf{w} \mathsf{R}{:}l_1.T_1 \oplus \ldots \oplus l_n.T_n \oplus l_\mathrm{d}$, where the latter case represents a weakly reliable branching prefix with the branches $l_1, \ldots, l_n, l_\mathrm{d}$, the default branch l_d, and where the next global type provides the missing specification for the default case.

We assume the sorts $\mathrm{S}_{\mathrm{belief}} = \{0, 1\}$ and $\mathrm{S}_{\mathrm{ack}} = \{\mathtt{t}, \mathtt{f}\}$. Let n be the number of agents. We start with the specification of the algorithm as a global type.

$$G_{rc}(\mathsf{n}) \triangleq (\mu t) \bigodot_{1\leq\mathsf{c}\leq\mathsf{n}} \left(\left(\bigodot_{1\leq i\leq\mathsf{n}, i\neq\mathsf{c}} i \to_\mathsf{u} \mathsf{c}{:}p_1\langle\mathrm{S}_{\mathrm{belief}}\rangle\right).\left(\bigodot_{1\leq i\leq\mathsf{n}, i\neq\mathsf{c}} \mathsf{c} \to_\mathsf{u} i{:}p_2\langle\mathrm{S}_{\mathrm{belief}}\rangle\right).\right.$$
$$\left(\bigodot_{1\leq i\leq\mathsf{n}, i\neq\mathsf{c}} i \to_\mathsf{u} \mathsf{c}{:}p_3\langle\mathrm{S}_{\mathrm{ack}}\rangle\right).$$
$$\left.\mathsf{c} \to_\mathsf{w} \{i | 1 \leq i \leq \mathsf{n}, i \neq \mathsf{c}\}{:}\mathit{Zero}.\mathsf{end} \oplus \mathit{One}.\mathsf{end} \oplus l_\mathrm{d}\right).t$$

It specifies a loop containing a collection of n rounds, where each process functions as a coordinator once. This collection of n rounds is specified with the first $\bigodot$, i.e., the continuation of l_d in the end of the description is the specification of round $\mathsf{r} + 1$ for all rounds $\mathsf{r} < \mathsf{n}$ whereas in the last round n we have $l_\mathrm{d}.t$. By unfolding the recursion on t, $G_{rc}(\mathsf{n})$ starts the next n rounds. The following three $\bigodot$ specify the Phases 1–3 of the algorithm within one round. Phase 4 is specified by a weakly reliable branching that does not need a $\bigodot$, since it is a broadcast.

In Phase 1 all processes except the coordinator c transmit a belief to c using label p_1. In Phase 2 c transmits a belief to all other processes using label p_2. Then all processes transmit a value of type $\mathrm{S}_{\mathrm{ack}}$ to the coordinator using label p_3 in Phase 3. Finally, in Phase 4 the coordinator broadcasts one of the labels Zero, One, or l_d, where the first two labels represent a decision and terminate the protocol, whereas the default label l_d specifies the need for another round. All interactions in the specification are unreliable or weakly reliable.

Let $\left(\bigodot_{1\leq i\leq n}\pi_i\right).P$ abbreviate the sequence $\pi_1.\ldots.\pi_n.P$, where $P \in \mathcal{P}$ is a process and $\pi_1, \ldots, \pi_n$ are sequences of prefixes.

$$Sys\left(\mathsf{n}, \vec{V}^0\right) \triangleq \overline{a}[\mathsf{n}](s).P\left(\mathsf{n}, \mathsf{n}, \vec{V}^0_\mathsf{n}\right) \mid \prod_{1\leq i<\mathsf{n}} a[i](s).P\left(i, \mathsf{n}, \vec{V}^0_i\right)$$
$$P\left(i, \mathsf{n}, \vec{V}\right) \triangleq (\mu X)\left(\bigodot_{1\leq\mathsf{c}\leq\mathsf{n}} \mathtt{if}\ i = \mathsf{c}\ \mathtt{then}\ P^\mathrm{C}_1\left(i, \mathsf{n}, \vec{V}\right)\ \mathtt{else}\ P^\mathrm{NC}_1\left(i, \mathsf{n}, \mathsf{c}, \vec{V}\right)\right).X$$

$Sys\left(\mathsf{n}, \vec{V}^0\right)$ describes the session initialisation of a system with n participants and the initial knowledge $\vec{V}^0$, where $\vec{V}^0_i$ is a vector that contains only the initial knowledge of role i. Let $\left|\vec{V}\right| \triangleq |\{i \mid v_i \neq \bot\}|$ return the number of non-empty entries. $P\left(i, \mathsf{n}, \vec{V}\right)$ describes a process i in a set of n processes. Each process then runs for at most n rounds and then loops.

$$P^\mathrm{C}_1\left(\mathsf{c}, \mathsf{n}, \vec{V}\right) \triangleq \left(\bigodot_{1\leq i\leq\mathsf{n}, i\neq\mathsf{c}} s[\mathsf{c}, i]?_\mathsf{u}p_1\langle\bot\rangle(v_i)\right).$$
$$\mathtt{if}\ \left|\vec{V}\right| \geq \left\lceil\frac{\mathsf{n} - 1}{2}\right\rceil\ \mathtt{then}\ P^\mathrm{C}_2\left(\mathsf{c}, \mathsf{n}, \mathrm{best}(\vec{V})\right)\ \mathtt{else}\ P^\mathrm{C}_2\left(\mathsf{c}, \mathsf{n}, \vec{V}^0_\mathsf{c}\right)$$

$$P_1^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V}\right) \triangleq s[\mathsf{i},\mathsf{c}]!_{\mathrm{u}}p_1\langle v_i\rangle.P_2^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V}\right)$$

Every non-coordinator $P_1^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V}\right)$ sends its own belief via unreliable communication to the coordinator and proceeds to Phase 2. The coordinator receives (some of) these messages and proceeds to Phase 2. If the reception of at least half of the messages was successful, it is updating its belief using the function best() that replaces all belief values with the best one. Otherwise, it discards all beliefs except its own. We are using $\left\lceil\frac{\mathsf{n}-1}{2}\right\rceil$ to check for a majority, since in our implementation processes do not transmit to themselves.

$$P_2^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right) \triangleq \Big(\bigodot_{1\leq\mathsf{i}\leq\mathsf{n},\mathsf{i}\neq\mathsf{c}} s[\mathsf{c},\mathsf{i}]!_{\mathrm{u}}p_2\langle v_i\rangle\Big).P_3^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right)$$

$$P_2^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V}\right) \triangleq s[\mathsf{i},\mathsf{c}]?_{\mathrm{u}}p_2\langle\perp\rangle(x).$$
$$\texttt{if } x = \perp \texttt{ then } P_3^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V},\mathtt{f}\right) \texttt{ else } P_3^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\mathrm{update}(\vec{V},\mathsf{i},x),\mathtt{t}\right)$$

In Phase 2, the coordinator sends its updated belief to all other processes via unreliable communication and proceeds. Note that, v_i is either $\perp$ for all $\mathsf{i}\neq\mathsf{c}$ or the best belief identified in Phase 1. If a non-coordinator process successfully receives a belief other than $\perp$, it updates its own belief with the received value and proceeds to Phase 3, where we use the Boolean value $\mathtt{t}$ for the acknowledgement. If the coordinator is suspected to have crashed or $\perp$ was received, the process proceeds to Phase 3 with the Boolean value $\mathtt{f}$, signalling nack.

$$P_3^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right) \triangleq \Big(\bigodot_{1\leq\mathsf{i}\leq\mathsf{n},\mathsf{i}\neq\mathsf{c}} s[\mathsf{c},\mathsf{i}]?_{\mathrm{u}}p_3\langle\perp\rangle(v_i)\Big).P_4^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right)$$

$$P_3^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V},b\right) \triangleq s[\mathsf{i},\mathsf{c}]!_{\mathrm{u}}p_3\langle b\rangle.P_4^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V}\right)$$

In Phase 3, every non-coordinator sends either ack or nack to the coordinator. If the coordinator successfully receives the message, it writes the Boolean value at the index of the sender into its knowledge vector. In case of failure, $\perp$ is used as default. After that the processes continue with Phase 4. Let $\mathcal{I} = \{\mathsf{i} \mid 1 \leq \mathsf{i} \leq \mathsf{n},\mathsf{i}\neq\mathsf{c}\}$.

$$P_4^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right) \triangleq \texttt{if } \mathrm{ack}(\vec{V}) \geq \left\lceil\frac{\mathsf{n}-1}{2}\right\rceil \texttt{ then } (\texttt{if } v_{\mathsf{c}} = 0 \texttt{ then } s[\mathsf{c},\mathcal{I}]!_{\mathrm{w}}\mathit{Zero}.\mathbf{0}$$
$$\texttt{else } s[\mathsf{c},\mathcal{I}]!_{\mathrm{w}}\mathit{One}.\mathbf{0}) \texttt{ else } s[\mathsf{c},\mathcal{I}]!_{\mathrm{w}}l_{\mathrm{d}}$$

$$P_4^{\mathrm{NC}}\left(\mathsf{i},\mathsf{n},\mathsf{c},\vec{V}\right) \triangleq s[\mathsf{i},\mathsf{n}]?_{\mathrm{w}}\mathit{Zero}.\mathbf{0} \oplus \mathit{One}.\mathbf{0} \oplus l_{\mathrm{d}}$$

In Phase 4 the coordinator checks if at least half of the non-coordinator roles signalled acknowledgement, utilising the function ack to count. If it received enough acknowledgements it transmits the decision via Zero or One and causes all participants to terminate. Otherwise, the coordinator sends the default label and continues with the next round. Remember that the missing continuation after the default label l_{d} for coordinators and non-coordinators is implemented by the next round.

We use a weakly reliable branching mechanism in conjunction with unreliable communication. The algorithm was modelled for systems with crash failures but without message loss. However, as long as the branching mechanism (i.e., the specified broadcast of decisions) is weakly reliable, we can relax the system requirements for the remainder of the algorithm. To ensure termination, however, we have to further restrict the number of lost messages.

6.2. **Failure Patterns.** Chandra and Toueg introduce in [CT96] also the failure detector $\Diamond\mathcal{S}$. The failure detector $\Diamond\mathcal{S}$ is called *eventually strong*, meaning that (1) eventually every process that crashes is permanently suspected by every correct process and (2) there is a time after which some correct process is never suspected by any other process. We observe that the suspicion of senders is only possible in Phase 3, where processes may suspect the coordinator of the round. Accordingly, the failure pattern FP_{uskip} implements this failure detector to allow processes to suspect unreliable coordinators in Phase 2, i.e., with label p_2. In Phase 1 and Phase 3 FP_{uskip} may allow to suspect processes that are not crashed after the coordinator received enough messages. In all other cases these two patterns eventually return true iff the respective sender is crashed. FP_{uget} can be used to reject outdated messages, since this is not important for this algorithm we implement it with the constant true. To ensure that messages of wrongly suspected coordinators in Phase 2 do not block the system, FP_{ml} is eventually true for messages with label p_2 that were suspected using $\Diamond\mathcal{S}$ or skipped p_1/p_3-messages and otherwise returns false. By the system requirements in [CT96], no messages get lost, but it is realistic to assume that receivers can drop messages of skipped receptions on their incoming message queues. As there are at least half of the processes required to be correct for this algorithm, we implement FP_{crash} by false if only half of the processes are alive and true otherwise. For the weakly reliable broadcast, FP_{wskip} returns true if and only if the respective coordinator is crashed, i.e., not suspected but indeed crashed. In [CT96] this broadcast, which is called just reliable in [CT96], is used to announce the decision. Since we use it for branching even before a decision was reached, our implementation is less efficient compared to [CT96]. We briefly discuss in the conclusions how to regain the original algorithm. These failure patterns satisfy the Condition 5.2.1–5.2.6.

6.3. **Termination, Agreement, and Validity.** Following [Tel94, Lyn96], a network of processes solves Consensus if
Termination: all non-failing participants eventually decide,
Agreement: all decision values are the same, and
Validity: each decision value is an initial value of some participant.
By [FMN07] the Rotating Coordinator algorithm solves Consensus in the presence of crash-failures.

In the proof of termination well-typedness of the implementation and progress in Theorem 5.10 are the main ingredients. Well-typedness ensures that the implementation follows its specification and progress ensures that it cannot get stuck. Apart from that, we only need to deduce from the system requirements, i.e., the used failure detector, that the implementation will eventually exit the loop.

Lemma 6.1. *The implementation of the rotating coordinator example satisfies termination.*

Proof. The global type ensures progress of the system, i.e., it either loops forever or terminates after a broadcast with *Zero* or *One*. Following the assumption of [CT96], there is eventually a round in which the coordinator is not suspected by any correct process and in which the coordinator does not suspect correct processes (or at least trusts a majority of them). Then the coordinator receives enough beliefs in Phase 1, transmits the best belief in Phase 2, receives enough acknowledgements in Phase 3, and transmits *Zero* or *One* in Phase 4. Finally, all correct processes receive the decision and terminate. Well-typedness and progress in Theorem 5.10 ensure that no process deadlocks along this way.　□

Validity can be checked by analysing the code of a single process and check whether it uses (to decide and to transmit) only its own initial belief or the belief it received from others. Session fidelity in Theorem 5.10 then ensures that there are no communication mismatches and all steps preserve this property.

Lemma 6.2. *The implementation of the rotating coordinator example satisfies validity.*

Proof. Well-typedness and the global type ensure that the communication structure is as specified. By the implementation, in Phase 1 every non-coordinator sends its own belief to the coordinator on label p_1 of which the coordinator selects the best value using the function best, thus altering its own belief with a valid value. By assumption, best does not modify any received belief value but only chooses one among them. Label p_2 is used to propagate the best belief of the coordinator of this round. On successful reception of this message, the non-coordinators update their own belief with the received value, which is a valid value. *Zero* and *One* are chosen matching to the current belief of the coordinator, i.e., the local belief is sent and updated on reception without alternation. As there are no more labels left on which belief values are sent, we conclude, that validity holds for each local process. Because of the global type and the communication protocol it ensures following session fidelity in Theorem 5.10—there are no communication mismatches and the received messages are of the expected sort—, then validity holds globally. □

Agreement follows from the weakly reliable broadcast, because the decision is broadcasted.

Lemma 6.3. *The implementation of the rotating coordinator example satisfies agreement.*

Proof. The only way to decide is via a broadcast. Let us fix a coordinator c of such a deciding round. To decide and broadcast a value, the coordinator c needs a majority of processes in its round. Messages $Zero^{\mathrm{w}}$ and One^{w} cannot be dropped and the failure pattern $\mathrm{FP}_{\mathtt{wskip}}$ does not allow to suspect a sender c of weakly reliable branching that is not crashed. Hence, a majority of processes receive the decision. Accordingly all decision values originate from the same broadcast. This ensures agreement. □

Combining the three above lemmata, we conclude that our implementation of the algorithm solves Consensus.

Theorem 6.4. *The algorithm solves Consensus, i.e., satisfies termination, validity, and agreement.*

Proof. By the Lemmata 6.1, 6.2, and 6.3. □

7. Conclusions

We present a fault-tolerant variant of MPST for systems that may suffer from message loss or crash failures. We implement unreliable communication and weakly reliable branching. The failure patterns in the semantics allow to verify algorithms modulo system requirements. We prove subject reduction and progress and present a small but relevant case study.

An open question is how to conveniently type unreliable recursive parts of protocols. Distributed algorithms are often recursive and exit this recursion if a result was successfully computed. We present a first attempt to solve this problem using a weakly reliable branching.

In further research we want to analyse, whether or in how far branching can be extended to the case of message loss.

Indeed our implementation of the rotating coordinator algorithm is not ideal. It implements the decision making procedure correctly and also allows processes to be in different rounds at the same time. So, it represents a non-trivial variant of the rotating coordinator algorithm. But it does not allow the processes to diverge in their rounds as freely as the original rotating coordinator algorithm, because the weakly reliable branching implementation implies that the coordinator has always to be the first process to leave a round. We can solve this problem by wrapping each round in an unreliable sub-session (e.g. an unreliable variant of the sub-sessions introduced in [DH12, Dem15]). If we allow processes to skip such an unreliable sub-session altogether, we obtain the intended behaviour.

We considered strongly reliable session delegation. Next we want to study whether and in how far we can introduce weakly reliable or unreliable session delegation. Similarly, we want to study unreliable variants of session initialisation including process crashes and lost messages during session initialisation. Moreover, unreliable variants of session initialisation open a new perspective on MPST-frameworks such as [CDA16] with dynamically changing network topologies and sessions for that the number of roles is determined at run-time.

In Section 5 we fix one set of conditions on failure patterns to prove subject reduction, session fidelity, and progress. We can also think of other sets of conditions. The failure pattern FP_{uget} can be used to reject the reception of outdated messages. Therefore, we drop Condition 5.2.2 and instead require for each message m whose reception is refused that FP_{ml} ensures that m is eventually dropped from the respective queue and that FP_{uskip} allows to skip the reception of these messages. An interesting question is to find minimal requirements and minimal sets of conditions that allow to prove correctness in general.

It would be nice to also fully automate the remaining proofs for the distributed algorithm in Section 6. The approach in [PWN19] sequentialises well-typed systems and gives the much simpler remaining verification problem to a model checker. Interestingly, the main challenges to adopt this approach are not the unreliable or weakly reliable prefixes but the failure patterns.

References

[ACT97]   Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In *Proc. of WDAG*, volume 1320 of *LNCS*, pages 126–140. Springer, 1997. `doi:10.1007/BFb0030680`.

[APN17]   Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session Types for Link Failures. In *Proc. of FORTE*, volume 10321 of *LNCS*, pages 1–16, 2017. `doi:10.1007/978-3-319-60225-7_1`.

[BCD+08]   Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, , and Nobuko Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. In *Proc. of CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008. `doi:10.1007/978-3-540-85361-9_33`.

[BHTY10]   Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In *Proc. of CONCUR*, volume 6269 of *LNCS*, pages 162–176. Springer, 2010. `doi:10.1007/978-3-642-15375-4_12`.

[CDA16]   Minas Charalambides, Peter Dinges, and Gul Agha. Parameterized, concurrent session types for asynchronous multi-actor interactions. *Science of Computer Programming*, 115–116:100–126, 2016. `doi:10.1016/j.scico.2015.10.006`.

[CDCG17]   Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Concurrent Reversible Sessions. In *Proc. of CONCUR*, volume 85 of *LIPIcs*, pages 30:1–30:17, 2017. `doi:10.4230/LIPIcs.CONCUR.2017.30`.

[CDCPY15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In *Proc. of SFM*, volume 9104 of *LNCS*, pages 146–178, 2015. `doi:10.1007/978-3-319-18941-3_4`.

[CDGH20] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. Global types with internal delegation. *Theoretical Computer Science*, 807:128–153, 2020. `doi:10.1016/j.tcs.2019.09.027`.

[CGY16] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016. `doi:10.1017/S0960129514000164`.

[CHY08] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Interactional Exceptions in Session Types. In *Proc. of CONCUR*, volume 5201 of *LNCS*, pages 402–417. Springer, 2008. `doi:10.1007/978-3-540-85361-9_32`.

[CS09] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Comput.*, 22(1):49–71, 2009. `doi:10.1007/s00446-009-0084-6`.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996. `doi:10.1145/226643.226647`.

[CV10] Luís Caires and Hugo Torres Vieira. Conversation types. *Theoretical Computer Science*, 411(51–52):4399–4440, 2010. `doi:10.1016/j.tcs.2010.09.010`.

[CVB+16] Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A Type Theory for Robust Failure Handling in Distributed Systems. In *Proc. of FORTE*, volume 9688 of *LNCS*, pages 96–113. Springer, 2016. `doi:10.1007/978-3-319-39570-8_7`.

[Dem15] R. Demangeon. Nested Protocols in Session Types. Personal communication about an extended version of [DH12] that is currently prepared by R. Demangeon., 2015.

[DH12] Romain Demangeon and Kohei Honda. Nested Protocols in Session Types. In *Proc. of CONCUR*, volume 7454 of *LNCS*, pages 272–286. Springer, 2012. `doi:10.1007/978-3-642-32940-1_20`.

[DHH+15] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods in System Design*, 46(3):197–225, 2015. `doi:10.1007/s10703-014-0218-8`.

[FH07] Adrian Francalanza and Matthew Hennessy. A fault tolerance bisimulation proof for consensus (extended abstract). In Rocco De Nicola, editor, *Proc. of ESOP*, volume 4421 of *LNCS*, pages 395–410. Springer, 2007. `doi:10.1007/978-3-540-71316-6_27`.

[FMN07] Rachele Fuzzati, Massimo Merro, and Uwe Nestmann. Distributed Consensus, revisited. *Acta Informatica*, pages 377–425, 2007. `doi:10.1007/s00236-007-0052-1`.

[Gär99] Felix C. Gärtner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1):1–26, 1999. `doi:10.1145/311531.311532`.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *Proc. of POPL*, volume 43, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

[HYC16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63(1), 2016. `doi:10.1145/2827695`.

[KGG14] Dimitrios Kouzapas, Ramūnas Gutkovas, and Simon J. Gay. Session Types for Broadcasting. In *Proc. of PLACES*, volume 155 of *EPTCS*, pages 25–31, 2014. `doi:10.4204/EPTCS.155.4`.

[KN09] Morten Kühnrich and Uwe Nestmann. On Process-Algebraic Proof Methods for Fault Tolerant Distributed Systems. In *Proc. of FORTE*, volume 5522 of *LNCS*, pages 198–212, 2009. `doi:10.1007/978-3-642-02138-1_13`.

[Lam01] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[NF03] Uwe Nestmann and Rachele Fuzzati. Unreliable failure detectors via operational semantics. In Vijay A. Saraswat, editor, *Proc. of ASIAN*, volume 2896 of *LNCS*, pages 54–71. Springer, 2003. `doi:10.1007/978-3-540-40965-6_5`.

[NFM03] Uwe Nestmann, Rachele Fuzzati, and Massimo Merro. Modeling consensus in a process calculus. In Roberto M. Amadio and Denis Lugiez, editors, *Proc. of CONCUR*, volume 2761 of *LNCS*, pages 393–407. Springer, 2003. `doi:10.1007/978-3-540-45187-7_26`.

[NY17] Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *Proc. of CC*, pages 98–108. ACM, 2017. `doi:10.1145/3033019.3033031`.

[PNW22a] Kirstin Peters, Uwe Nestmann, and Christoph Wagner. Fault-Tolerant Multiparty Session Types. In *Proc. of Forte*, volume 13273 of *LNCS*, pages 93–113. Springer, 2022. `doi:10.1007/978-3-031-08679-3_7`.

[PNW22b] Kirstin Peters, Uwe Nestmann, and Christoph Wagner. Fault-Tolerant Multiparty Session Types (Technical Report). Technical report, 2022. `doi:10.48550/arXiv.2204.07728`.

[PWN19] Kirstin Peters, Christoph Wagner, and Uwe Nestmann. Taming Concurrency for Verification Using Multiparty Session Types. In *Proc. of ICTAC*, volume 11884 of *LNCS*, pages 196–215, 2019. `doi:10.1007/978-3-030-32505-3_12`.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[VCE+18] Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In *Proc. of ESOP*, volume 10801 of *LNCS*, pages 799–826. Springer, 2018. `doi:10.1007/978-3-319-89884-1_28`.

[vGHH21] Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom. In *Proc. of LICS*, pages 1–13. IEEE, 2021.

[vST17] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 2017.

[WN14] Christoph Wagner and Uwe Nestmann. States in Process Calculi. In *Proc. of EXPRESS/SOS*, volume 160 of *EPTCS*, pages 48–62, 2014. `doi:10.4204/EPTCS.160.6`.

[YDBH10] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised Multiparty Session Types. In *Proc. of FoSSaCS*, volume 6014 of *LNCS*, pages 128–145, 2010. `doi:10.1007/978-3-642-12032-9_10`.