UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Fault-tolerant multiparty session types with global escape loops

Lukas Bartl, Julian Linne, Kirstin Peters

Angaben zur Veröffentlichung / Publication details:

Bartl, Lukas, Julian Linne, and Kirstin Peters. 2025. "Fault-tolerant multiparty session types with global escape loops." *Electronic Proceedings in Theoretical Computer Science* 433: 3–22. https://doi.org/10.4204/eptcs.433.3.

Nutzungsbedingungen / Terms of use:

CC BY 4.0

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
CC-BY 4.0: Creative Commons: Namensnennung
Weitere Informationen finden Sie unter: / For more information see:
https://creativecommons.org/licenses/by/4.0/deed.de

Fault-Tolerant Multiparty Session Types with Global Escape Loops

Lukas Bartl Julian Linne Kirstin Peters
Universität Augsburg, Germany

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 FTMPST—a version of fault-tolerant multiparty session types with failure patterns to represent system requirements for system failures such as unreliable communication and process crashes—by a novel, fault-tolerant loop construct with global escapes that does not require global coordination. Each process runs its own local version of the loop. If a process finds a solution to the considered problem, it does not only terminate its own loop but also informs the other participants via `exit`-messages. Upon receiving an `exit`-message, a process immediately terminates its algorithm. To increase efficiency and model standard fault-tolerant algorithms, these messages are non-blocking, i.e., a process may continue until a possibly delayed `exit`-message is received. 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 [19, 14]. 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 [19, 4, 5]. 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 [22, 28], 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. We focus on masking fault-tolerant algorithms (see [17]), i.e., safety and liveness requirements hold despite failures without further intervention by the programmer.

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. [8, 7, 13, 29, 15, 1]). These approaches are sometimes enhanced with recovery mechanisms such as [9] or even provide algorithms to help find safe states to recover from, as in [23]. Many of these approaches introduce nested TRY-and-CATCH-blocks and a challenge is to

C. Di Giusto and G. Bacci (Eds.): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2025)
EPTCS 433, 2025, pp. 3–22, doi:10.4204/EPTCS.433.3

© L. Bartl, J. Linne, & K. Peters
This work is licensed under the Creative Commons Attribution License.

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.

Type systems are usually designed for failure-free scenarios. An exception is [20] 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 [8, 7, 13, 29, 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. Here we deliberately do not model 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 [7], 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; one of their most challenging problems. *Distributed* processes usually cannot reliably observe an error on another system part, unless they are informed by some system "device" (like the "coordinator" of [29] or the "oracle" of [7]). Therefore, abstractions like unreliable failure detectors are used to model this restricted observability which can, for example, be implemented by time-outs.

We extend FTMPST [24, 25], a version of fault-tolerant multiparty session types with failure patterns to represent system requirements for system failures such as unreliable communication and process crashes. We add a novel, fault-tolerant loop construct with global escapes but without a need for global coordination. Thereby, we tackle an open question of [25], namely how to conveniently type unreliable recursive parts of protocols. Distributed algorithms are often recursive and exit this recursion if a result was successfully computed. In [25], weakly reliable branching was used to exit a standard recursion. Unfortunately, this operation temporarily blocks some processes. Our novel loop construct overcomes this problem.

Each loop of an algorithm has a unique identifier, where unique means from a global point of view. Each process runs its own local version of the loop, but the local loops that jointly define a recursive routine of the algorithm have the same identifier. If a process finds a solution to the considered problem, it does not only terminate its own loop but also informs the other participants via `exit`-messages that may carry a solution value. Upon receiving an `exit`-message, a process immediately terminates its algorithm. To increase efficiency and model standard fault-tolerant algorithms, these messages are non-blocking, i.e., a process may continue until a possibly delayed `exit`-message is received. Since communication in the system is asynchronous and because of faults such as message delays, many algorithms do not forbid that different participants terminate the protocol concurrently. Hence, there may be several concurrent `exit`-messages for the same local loop. The algorithm then has to ensure, that all of them carry the same solution value—usually called *agreement*.

To guide the behaviour of unreliable communication, we inherit from [25] the *failure patterns* used in the semantics of processes. Note that these patterns 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. To illustrate our approach we analyse a variant of the well-known rotating coordinator algorithm by Chandra and Toueg.

Additional material and the missing proofs are contained in a technical report [3].

2 Fault-Tolerant Types and Processes

Following [25], 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 [2] 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. Our new loop construct relies on unreliable interactions for the loop body such that the termination of the loop does not cause any blocking of the interaction partners. However, the `exit`-messages should not be dropped before the loop is terminated and are thus weakly reliable.

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 $\mathscr{N}$ of names $a,s,x\ldots$; $\mathscr{R}$ of roles $\mathsf{n},\mathsf{r},\ldots$; $\mathscr{L}$ of labels $l,l_{\mathrm{d}},\ldots$; $\mathscr{V}_{\mathrm{T}}$ of type variables t; and $\mathscr{V}_{\mathrm{P}}$ of process variables X are pairwise distinct. To simplify the reduction semantics of our session calculus, we use natural numbers as roles (compare to [19]). Sorts S range over $\mathbb{B},\mathbb{N},\ldots$. The set $\mathscr{E}$ of expressions $e,v,b,\ldots$ is constructed from the standard Boolean operations, natural numbers, standard arithmetic operators, tuples, names, and (in)equalities. We assume an evaluation function $\mathrm{eval}(\cdot)$ that evaluates expressions to values.

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 start from standard MPST [18, 19] extended by unreliable communication and weakly reliable branching in [24, 25]. We then add an unreliable loop construct with weakly reliable global escapes (highlighted in blue) in Figure 1.

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

The type $\mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\langle \mathrm{S}\rangle.G$ specifies a strongly reliable communication from role r_1 to role r_2 to transmit a value of sort S and then continues with G. A system with this 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 a 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 sort S and then continues (regardless of the success of this communication) with

Global Types	Local Types	Processes
		$P ::= \overline{a}[\mathsf{n}](s).P \mid a[\mathsf{r}](s).P$
$G ::= \mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\langle \mathrm{S}\rangle.G$	$T ::= [\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T$	$\mid s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}\langle e\rangle.P$
	$\mid [\mathsf{r}_1]?_{\mathrm{r}}\langle \mathrm{S}\rangle.T$	$\mid s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{r}}(x).P$
$\mid \mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S}\rangle.G$	$\mid [\mathsf{r}_2]!_{\mathrm{u}}l\langle \mathrm{S}\rangle.T$	$\mid s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{u}}l\langle e\rangle.P$
	$\mid [\mathsf{r}_1]?_{\mathrm{u}}l\langle \mathrm{S}\rangle.T$	$\mid s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{u}}l\langle v\rangle(x).P$
$\mid \mathsf{r}_1 \to_{\mathrm{r}} \mathsf{r}_2{:}\{l_i.G_i\}_{i\in \mathrm{I}}$	$\mid [\mathsf{r}_2]!_{\mathrm{r}}\{l_i.T_i\}_{i\in \mathrm{I}}$	$\mid s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathrm{r}}l.P$
	$\mid [\mathsf{r}_1]?_{\mathrm{r}}\{l_i.T_i\}_{i\in \mathrm{I}}$	$\mid s[\mathsf{r}_2,\mathsf{r}_1]?_{\mathrm{r}}\{l_i.P_i\}_{i\in \mathrm{I}}$
$\mid \mathsf{r} \to_{\mathrm{w}} \mathsf{R}{:}\{l_i.G_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$	$\mid [\mathsf{R}]!_{\mathrm{w}}\{l_i.T_i\}_{i\in \mathrm{I}}$	$\mid s[\mathsf{r},\mathsf{R}]!_{\mathrm{w}}l.P$
	$\mid [\mathsf{r}]?_{\mathrm{w}}\{l_i.T_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$	$\mid s[\mathsf{r}_j,\mathsf{r}]?_{\mathrm{w}}\{l_i.P_i\}_{i\in \mathrm{I},l_{\mathrm{d}}}$
$\mid G_1 \mid\mid G_2$		$\mid P_1 \mid P_2$
$\mid (\mu t,\mathsf{c})G \mid t$	$\mid (\mu t,\mathsf{c}=n)T \mid t$	$\mid (\mu X,\mathsf{c}=n)P \mid X$
$\mid \mathtt{end}$	$\mid \mathtt{end}$	$\mid \mathbf{0}$
$\mid [\mathsf{R}]\infty_e^{\mathsf{c}}\langle \mathrm{S}_0\rangle.G_0;\langle \mathrm{S}_2\rangle.G_2$	$\mid [\mathsf{R}]\infty_e^{\mathsf{c}=n}[\langle \mathrm{S}_0\rangle.T_0]T_1;\langle \mathrm{S}_2\rangle.T_2$	$\mid s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1;(y).P_2$
$\mid \mathtt{call}\langle e\rangle$	$\mid \mathtt{call}\langle e\rangle$	$\mid \mathtt{call}\langle e,e'\rangle \mid \mathtt{exit}\langle e,e'\rangle$
		$\mid \mathtt{if}\ b\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2$
		$\mid (\nu x)P \mid \bot$
$\mid \mathsf{r}_1 \to \mathsf{r}_2{:}\langle s'[\mathsf{r}]{:}T\rangle.G$	$\mid [\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T\rangle.T'$	$\mid s[\mathsf{r}_1,\mathsf{r}_2]!\langle\langle s'[\mathsf{r}]\rangle\rangle.P$
	$\mid [\mathsf{r}_1]?\langle s'[\mathsf{r}]{:}T\rangle.T'$	$\mid s[\mathsf{r}_2,\mathsf{r}_1]?((s'[\mathsf{r}])).P$
		$\mid s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}$

Message Types	Messages
$\mathsf{mt} ::= \langle \mathrm{S}\rangle^{\mathrm{r}} \mid l\langle \mathrm{S}\rangle^{\mathrm{u}} \mid l^{\mathrm{r}} \mid l^{\mathrm{w}}$	$\mathsf{m} ::= \langle v\rangle^{\mathrm{r}} \mid l\langle v\rangle^{\mathrm{u}} \mid l^{\mathrm{r}} \mid l^{\mathrm{w}}$
$\mid \mathtt{exit}\langle id,\mathrm{S}\rangle \mid s[\mathsf{r}]$	$\mid \mathtt{exit}\langle id,v\rangle \mid s[\mathsf{r}]$
$\mathsf{MT} ::= [] \mid \mathsf{mt}\#\mathsf{MT}$	$\mathrm{M} ::= [] \mid \mathsf{m}\#\mathrm{M}$

Figure 1: Syntax of Fault-Tolerant MPST with Global Escape Loops.

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 does not influence 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 [25], 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 $\mathsf{R} \subseteq \mathscr{R}$ 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 (see [25] for an explanation). The type system ensures that all processes that are not crashed will move to the same branch. We often abbreviate branching w.r.t. 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}.

We extend the standard operators for recursion $(\mu t)G$, $(\mu t)T$, and $(\mu X)P$ of [25] by a counter $(\mu X,\mathsf{c}=n)P$ (specified as $(\mu t,\mathsf{c}=n)T$ with the global type $(\mu t,\mathsf{c})G$), where n is a natural number that is increased by unfolding recursion and c can be used as pointer to the current value of the counter within

expressions in G, T, and P. These expressions allow us to construct unique identifiers for loops within a surrounding recursion.

A loop $s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1;(y).P_2$ (specified as $[\mathsf{R}]\infty_e^{\mathsf{c}=n}[\langle \mathsf{S}_0\rangle.T_0]T_1;\langle \mathsf{S}_2\rangle.T_2$ with the global type $[\mathsf{R}]\infty_e^{\mathsf{c}}\langle \mathsf{S}_0\rangle.G_0;\langle \mathsf{S}_2\rangle.G_2$) creates a loop in that role r of session s is currently running the *loop body* P_1 and may interact with the roles in R that are running their local versions of this loop. We identify a loop with its unique identifier $id = \mathrm{eval}(e)$ that is unique for the whole derivation of the system and the same for all roles $\mathsf{R}\cup\{\mathsf{r}\}$. Again, the loop has a counter $\mathsf{c} = n$ that is increased in unfolding loops and can be used to create the unique identifiers of loops nested within the current loop. Communication within a loop is unreliable. With $\mathtt{call}\langle id, v\rangle$ (specified as $\mathtt{call}\langle id\rangle$) role r invokes another iteration of the loop id given by the *loop program* $(x).P_0$, where x is instantiated with v. Role r can terminate its own loop and all loops of the other R by sending $\mathtt{exit}\langle id, v'\rangle$. In this case, or upon receiving $\mathtt{exit}\langle id, v'\rangle$, role r continues with the *loop continuation* $(y).P_2$ of loop $id = \mathrm{eval}(e)$, where y is instantiated by v'. The loop body P_1 contains whatever is left of the current iteration of the loop program P_0. We initialise, as expected by the type system, a loop as $s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=0}[(x).P_0]\mathtt{call}\langle e, v\rangle;(y).P_2$ such that its first step calls the first iteration of the loop.

The $\perp$ denotes a process that crashed. Similar to [19], we use message queues to implement asynchrony in sessions. Therefore, session initialisation introduces a directed and initially empty message queue $s_{\mathsf{r}_i\to\mathsf{r}_j}{:}[\,]$ for each pair of roles $\mathsf{r}_i \neq \mathsf{r}_j$ 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 makes sense to further split the message queues into three message queues for each pair $\mathsf{r}_i \neq \mathsf{r}_j$ 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 $\mathsf{r}_i \neq \mathsf{r}_j$. However, the correctness of our type system does not depend on this decision. We have six kinds of messages m and corresponding message types mt in Figure 1—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 $\langle \mathrm{S}\rangle^{\mathrm{r}}$. In unreliable communication, the message $l\langle v\rangle^{\mathrm{u}}$ (of type $l\langle \mathrm{S}\rangle^{\mathrm{u}}$) 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. The message $\mathtt{exit}\langle id, v\rangle$ of type $\mathtt{exit}\langle id, \mathrm{S}\rangle$ signals that the loop id can be terminated. Finally, the 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 $G \,||\, G'$; parallel composition $P \mid P'$; inaction $\mathtt{end}$, $\mathbf{0}$; 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 and, similarly, that recursive calls $\mathtt{call}\langle e, e'\rangle, \mathtt{call}\langle e\rangle$ are guarded within loop programs and do not occur outside of the declaration of loop $\mathrm{eval}(e)$ in types or processes. To ensure that loops are uniquely identified, their identifiers are described as expressions that have to evaluate to a unique identifier in a type and all its unfoldings of recursion. That is to say, within standard recursion or surrounding loops, these identifiers have to be built by a mechanism that ensures uniqueness, such as the counter of the surrounding recursion. More precisely, all iterations of a loop have the same identifier, whereas a loop within a surrounding recursion or loop needs a fresh identifier for every iteration of the surrounding recursion or loop. Moreover, the type system ensures that neither loop bodies nor loop programs may contain free type variables.

In types $(\mu t, \mathsf{c})G$ and $(\mu t, \mathsf{c} = n)T$ the type variable t and the variable c are *bound* in G, T. In processes $(\mu X, \mathsf{c} = n)P$ the process variable X and the variable c are bound in P. Similarly, in loops $[\mathsf{R}]\infty_e^{\mathsf{c}}\langle \mathsf{S}_0\rangle.G_0;\langle \mathsf{S}_2\rangle.G_2$, $[\mathsf{R}]\infty_e^{\mathsf{c}=n}[\langle \mathsf{S}_0\rangle.T_0]T_1;\langle \mathsf{S}_2\rangle.T_2$, and $s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1;(y).P_2$ the variable c is

bound in G_0, T_0, and P_0. Additionally, 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 $s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathsf{r}}(x).P$. 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 '.'. Moreover, a loop is a guard for its loop continuation, but its loop body is unguarded. 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, G, or T do not contain message queues. We write $\mathrm{unr}(A)$ if $\mathrm{nsr}(A)$ and none of the prefixes in A is a weakly reliable branching.

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

As discussed in [25], labels may carry additional runtime information such as timestamps, in order to provide the technical means to implement the failure patterns introduced with the semantics below.

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 $l_P \doteq l_T \wedge l_T \not\doteq l'_T \Rightarrow l_P \not\doteq l'_T$.

Of course, the presented type system remains valid if we use labels without additional runtime information. 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. Similarly, labels are used in [6] to avoid communication errors.

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*.

Projection maps global types onto the respective local type for a given role p. Recursion and loops are projected as follows:

$$((\mu t, \mathsf{c})G)\upharpoonright_{\mathsf{p}} \triangleq \begin{cases} G\{^0/_{\mathsf{c}}\}\upharpoonright_{\mathsf{p}} & \text{if } t \text{ does not occur in } G \\ (\mu t, \mathsf{c}=0)G\upharpoonright_{\mathsf{p}} & \text{else if } \mathsf{p} \in \mathrm{R}(G) \\ \mathtt{end} & \text{otherwise} \end{cases}$$

$$([\mathsf{R}]\infty_e^{\mathsf{c}}\langle \mathrm{S}_0\rangle.G_0;\langle \mathrm{S}_2\rangle.G_2)\upharpoonright_{\mathsf{p}} \triangleq \begin{cases} [\mathsf{R}\setminus\{\mathsf{p}\}]\infty_e^{\mathsf{c}=0}[\langle \mathrm{S}_0\rangle.G_0\upharpoonright_{\mathsf{p}}]\mathtt{call}\langle e\rangle;\langle \mathrm{S}_2\rangle.G_2\upharpoonright_{\mathsf{p}} & \text{if } \mathsf{p}\in\mathsf{R} \\ G_2\upharpoonright_{\mathsf{p}} & \text{otherwise} \end{cases}$$

Projection of recursion is standard except for the initialisation of the counter c with 0. Recursive types without their recursion variable are mapped to the projection of their recursion body (similar to [10]), else if p occurs in the recursion body we map to a recursive local type, or else to successful termination. If projected on one of its roles $\mathsf{p} \in \mathsf{R}$, the global specification of the loop program G_0 and the global specification of the loop continuation G_2 are projected on p. The counter is initialised with 0 and the loop body is instantiated with $\texttt{call}\langle e \rangle$ to call the first loop iteration. Else, the loop is skipped and we project the loop continuation G_2 on p.

Projection of the remaining operators is given in [25]. We restrict our attention to projectable and well-formed types, as defined in [3].

3 A Semantics with Failure Patterns for Global Escape Loops

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 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:

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

For the reduction semantics in Figure 2 we start with the rules of fault-tolerant processes from [25] that we extend with the rules for our new loops (in blue colour). Similar to [19], session initialisation is synchronous and communication within a session is asynchronous using message queues. The rules are standard except for the six failure patterns (five patterns from [25] and one new pattern $\texttt{FP}_{\texttt{drop}}$ for loops) and three rules for system failures: (Crash) for *crash failures*, (ML) for *message loss*, and the new rule (EDrop) that allows to drop `exit`-messages of loops. *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 $v = \mathrm{eval}(e)$ 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 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 v.

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 $\texttt{FP}_{\texttt{uget}}$ to decide whether this step is allowed. This failure pattern could, e.g., be used to reject messages that are too old. The condition $l \doteq l'$ ensures that the static information in the transmitted label matches the expectation specified in the label of the receiver to avoid communication mismatches. The Rule (USkip) allows to skip

$$
\begin{array}{lll}
\text{(Init)} & \overline{a}[\mathsf{n}](s).P_{\mathsf{n}} \mid \prod_{1\le \mathsf{i}\le \mathsf{n}-1} a[\mathsf{i}](s).P_{\mathsf{i}} \longmapsto (\nu s)\left(\prod_{1\le \mathsf{i}\le \mathsf{n}} P_{\mathsf{i}} \mid \prod_{1\le \mathsf{i},\mathsf{j}\le \mathsf{n}, \mathsf{i}\neq \mathsf{j}} s_{\mathsf{i}\to\mathsf{j}}{:}[\,]\right) & \text{if } a \neq s \\
\text{(RSend)} & s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathsf{r}}\langle e\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^{\mathsf{r}} & \text{if } \mathrm{eval}(e) = v \\
\text{(RGet)} & 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} \longmapsto P\{{}^{v}/_{x}\} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M} & \\
\text{(USend)} & s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathsf{u}} l\langle e\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^{\mathsf{u}} & \text{if } \mathrm{eval}(e) = v \\
\text{(UGet)} & 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} \longmapsto P\{{}^{v}/_{x}\} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M} & \text{if } l \doteq l',\ \mathtt{FP}_{\mathtt{uget}}(s,\mathsf{r}_1,\mathsf{r}_2,l',\ldots) \\
\text{(USkip)} & s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathsf{u}} l\langle dv\rangle(x).P \longmapsto P\{{}^{dv}/_{x}\} & \text{if } \mathtt{FP}_{\mathtt{uskip}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots) \\
\text{(ML)} & s_{\mathsf{r}_1\to\mathsf{r}_2}{:}l\langle v\rangle^{\mathsf{u}}\#\mathrm{M} \longmapsto s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M} & \text{if } \mathtt{FP}_{\mathtt{ml}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots) \\
\text{(RSel)} & s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathsf{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^{\mathsf{r}} & \\
\text{(RBran)} & s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathsf{r}}\{l_i.P_i\}_{i\in \mathrm{I}} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}l^{\mathsf{r}}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M} & \text{if } l \doteq l_j,\ j \in \mathrm{I} \\
\text{(WSel)} & s[\mathsf{r},\mathsf{R}]!_{\mathsf{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^{\mathsf{w}} & \\
\text{(WBran)} & s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathsf{w}}\{l_i.P_i\}_{i\in \mathrm{I},l_{\mathrm{d}}} \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}l^{\mathsf{w}}\#\mathrm{M} \longmapsto P_j \mid s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M} & \text{if } l \doteq l_j,\ j \in \mathrm{I} \\
\text{(WSkip)} & s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathsf{w}}\{l_i.P_i\}_{i\in \mathrm{I},l_{\mathrm{d}}} \longmapsto P_{\mathrm{d}} & \text{if } \mathtt{FP}_{\mathtt{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2,\ldots) \\
\text{(LStep)} & s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1;(y).P_2 \mid Q \longmapsto s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1';(y).P_2 \mid Q' & \\
 & & \text{if } P_1 \mid Q \longmapsto P_1' \mid Q',\ \mathtt{onlyMQ}_{\mathsf{r}\leftrightarrow\mathsf{R}}(Q,Q') \\
\text{(LCall)} & s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]\mathtt{call}\langle e_l,e_v\rangle;(y).P_2 \longmapsto & \\
 & s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=\mathrm{eval}(n+1)}[(x).P_0]\,(P_0\{{}^{n}/_{\mathsf{c}}\})\,\{{}^{v}/_{x}\};(y).P_2 & \text{if } \mathrm{eval}(e) = \mathrm{eval}(e_l),\ \mathrm{eval}(e_v) = v \\
\text{(LExitS)} & s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]\mathtt{exit}\langle e_l,e_v\rangle;(y).P_2 \mid \prod_{\mathsf{r}_i\in\mathsf{R}} s_{\mathsf{r}\to\mathsf{r}_i}{:}\mathrm{M}_i \longmapsto & \\
 & P_2\{{}^{v}/_{y}\} \mid \prod_{\mathsf{r}_i\in\mathsf{R}} s_{\mathsf{r}\to\mathsf{r}_i}{:}\mathrm{M}_i\#\mathtt{exit}\langle id,v\rangle & \text{if } \mathrm{eval}(e) = \mathrm{eval}(e_l) = id,\ \mathrm{eval}(e_v) = v \\
\text{(LExitG)} & s[\mathsf{r},\mathsf{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1;(y).P_2 \mid s_{\mathsf{r}'\to\mathsf{r}}{:}\mathtt{exit}\langle id,v\rangle\#\mathrm{M} \longmapsto & \\
 & P_2\{{}^{v}/_{y}\} \mid s_{\mathsf{r}'\to\mathsf{r}}{:}\mathrm{M} & \text{if } \mathrm{eval}(e) = id,\ \mathsf{r}' \in \mathsf{R} \\
\text{(EDrop)} & s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathtt{exit}\langle id,v\rangle\#\mathrm{M} \longmapsto s_{\mathsf{r}_2\to\mathsf{r}_1}{:}\mathrm{M} & \text{if } \mathtt{FP}_{\mathtt{drop}}(\mathsf{r},id) \\
\text{(Crash)} & P \longmapsto \bot & \text{if } \mathtt{FP}_{\mathtt{crash}}(P,\ldots) \\
\text{(If-T)} & \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \longmapsto P & \text{if } \mathrm{eval}(e) \text{ is true} \\
\text{(If-F)} & \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \longmapsto P' & \text{if } \mathrm{eval}(e) \text{ is false} \\
\text{(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}] & \\
\text{(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} & \\
\text{(Par)} & P_1 \mid P_2 \longmapsto P_1' \mid P_2 & \text{if } P_1 \longmapsto P_1' \\
\text{(Res)} & (\nu x)P \longmapsto (\nu x)P' & \text{if } P \longmapsto P' \\
\text{(Rec)} & (\mu X,\mathsf{c}=n)P \longmapsto (P\{{}^{n}/_{\mathsf{c}}\})\,\{{}^{(\mu X,\mathsf{c}=\mathrm{eval}(n+1))P}/_{X}\} & \\
\text{(Struc)} & P_1 \longmapsto P_1' & \text{if } P_1 \equiv P_2,\ P_2 \longmapsto P_2',\ P_2' \equiv P_1'
\end{array}
$$

Figure 2: Reduction Rules ($\longmapsto$) of Fault-Tolerant Processes with Global Escape Loops.

the reception of a message in an unreliable communication using a failure pattern $\mathtt{FP}_{\mathtt{uskip}}$ and instead substitutes the bound variable x in the continuation with the default value dv. The failure pattern $\mathtt{FP}_{\mathtt{uskip}}$ 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 $\mathtt{FP}_{\mathtt{wskip}}$ 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.

With (LStep) the body of a loop may (1) send a message to a message queue, (2) receive a message from a queue, (3) resolve a conditional, or (4) skip an outer loop-construct of nested loops to perform

an output, input, call another loop iteration, or exit a loop. Therefore, the predicate $\texttt{onlyMQ}_{\mathsf{r}\leftrightarrow\mathsf{R}}(Q,Q')$ checks that Q and Q' consist only of message queues from r into roles within R or the other way around. Rule (LCall) puts loop eval(e) onto another iteration, where c is replaced by the current counter value n and x is instantiated with eval$(e_v) = v$ in the loop program $(x).P_0$. Additionally, the counter is increased by 1. The side condition eval$(e) =$ eval(e_l) ensures that the correct loop is iterated. Role r can terminate its loop eval$(e) =$ eval$(e_l) = id$ with (LExitS). This step reduces r to its loop continuation $(y).P_2$, where y is instantiated with eval$(e_v) = v$. It then adds the message $\texttt{exit}\langle id, v\rangle$ to the message queues from r to all roles in R. Upon receiving $\texttt{exit}\langle id, v\rangle$ in rule (LExitG), role r is induced to also terminate its loop id and continue with its loop continuation instantiated with v.

The Rules (Crash) for *crash failures* and (ML) for *message loss*, describe failures of a system. With Rule (Crash), P can crash if $\texttt{FP}_{\texttt{crash}}$, where $\texttt{FP}_{\texttt{crash}}$ can e.g. model immortal processes or global bounds on the number of crashes. (ML) allows to drop an unreliable message if the failure pattern $\texttt{FP}_{\texttt{ml}}$ is valid. $\texttt{FP}_{\texttt{ml}}$ allows, e.g., to implement safe channels that never lose messages or a global bound on the number of lost messages. Rule (EDrop), similarly allows to *drop* a message from a queue, but it does not implement a failure. Instead it allows us to drop `exit`-messages of already terminated loops, i.e., $\texttt{FP}_{\texttt{drop}}$ checks whether the loop mentioned by the `exit`-message of the considered role is already terminated and only in this case allows to drop the message. Since a loop id is run concurrently by several roles of which each role runs its local version of the loop id, it cannot be avoided that several roles may actively terminate their loop concurrently, causing several `exit`-messages for the same loop.

The remaining reduction rules for conditionals, delegation, parallel composition, restriction, recursion, and structural congruence are standard, except for the counter in unfolding recursion.

We deliberately do not specify failure patterns, although we usually assume that the failure patterns $\texttt{FP}_{\texttt{uget}}$, $\texttt{FP}_{\texttt{uskip}}$, $\texttt{FP}_{\texttt{wskip}}$, and $\texttt{FP}_{\texttt{drop}}$ use only local information, whereas $\texttt{FP}_{\texttt{ml}}$ and $\texttt{FP}_{\texttt{crash}}$ 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 1.1–1.8 in Section 4 and explicitly in Section 5. If we instantiate the patterns $\texttt{FP}_{\texttt{uget}}$ with true, the patterns $\texttt{FP}_{\texttt{uskip}}$, $\texttt{FP}_{\texttt{wskip}}$, $\texttt{FP}_{\texttt{crash}}$, $\texttt{FP}_{\texttt{ml}}$ with false, and the pattern $\texttt{FP}_{\texttt{drop}}$ such that it is true whenever the mentioned loop is terminated by the mentioned role, then we obtain a system without failures. In contrast, the instantiation of $\texttt{FP}_{\texttt{drop}}$ as above and the other 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, $\texttt{FP}_{\texttt{ml}}$ and $\texttt{FP}_{\texttt{crash}}$ are thus often implemented as oracles for verification, whereas e.g. $\texttt{FP}_{\texttt{uskip}}$ and $\texttt{FP}_{\texttt{wskip}}$ 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, processes, and loop-identifiers mentioned in Figure 2 are not parameters of the failure patterns, but just a vehicle to more formally specify the conditions on failure patterns in Section 4. 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 2 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.

The following toy-example illustrates nested loops in types and projection. A more interesting example with communication is given in Sections 5.

$$G \triangleq (\mu t, \mathtt{c}_1)[\{1\}]\infty^{\mathtt{c}_2}_{\mathtt{c}_1}\langle\mathbb{N}\rangle.[\{1\}]\infty^{\mathtt{c}_3}_{(\mathtt{c}_1,\mathtt{c}_2)}\langle\mathbb{N}\rangle.\mathtt{end};\langle\mathbb{N}\rangle.\mathtt{call}\langle\mathtt{c}_1\rangle;\langle\mathbb{N}\rangle.t$$

$$G\upharpoonright_1 = (\mu t, \mathtt{c}_1 = 0)[\emptyset]\infty^{\mathtt{c}_2=0}_{\mathtt{c}_1}\Big[\langle\mathbb{N}\rangle.[\emptyset]\infty^{\mathtt{c}_3=0}_{(\mathtt{c}_1,\mathtt{c}_2)}[\langle\mathbb{N}\rangle.\mathtt{end}]\mathtt{call}\langle(\mathtt{c}_1,\mathtt{c}_2)\rangle;\langle\mathbb{N}\rangle.\mathtt{call}\langle\mathtt{c}_1\rangle\Big]\mathtt{call}\langle\mathtt{c}_1\rangle;\langle\mathbb{N}\rangle.t$$

To ensure that the loops are uniquely identified in all unfoldings of the surrounding recursion and the outer loop, their identifiers $\mathtt{c}_1$ and $(\mathtt{c}_1,\mathtt{c}_2)$ are build from counters. This type can be implemented as:

$$\begin{aligned}
P &\triangleq (\mu X, \mathtt{c}_1 = 0)P_{\mathtt{c}_1}\\
P_{\mathtt{c}_1} &\triangleq s[1,\emptyset]\infty^{\mathtt{c}_2=0}_{\mathtt{c}_1}[(x).P_{\mathtt{c}_1,\mathtt{c}_2}(x)]\mathtt{call}\langle\mathtt{c}_1,0\rangle;(y).X\\
P_{\mathtt{c}_1,\mathtt{c}_2}(x) &\triangleq s[1,\emptyset]\infty^{\mathtt{c}_3=0}_{(\mathtt{c}_1,\mathtt{c}_2)}\left[\left(x'\right).\mathtt{exit}\left\langle(\mathtt{c}_1,\mathtt{c}_2),x'+1\right\rangle\right]\mathtt{call}\left\langle(\mathtt{c}_1,\mathtt{c}_2),x+1\right\rangle;\left(y'\right).P_{\mathtt{c}_1,\mathtt{cont}}(y')\\
P_{\mathtt{c}_1,\mathtt{cont}}(y') &\triangleq \mathtt{if}\ y'<5\ \mathtt{then}\ \mathtt{call}\left\langle\mathtt{c}_1,y'+1\right\rangle\ \mathtt{else}\ \mathtt{exit}\left\langle\mathtt{c}_1,y'+1\right\rangle
\end{aligned}$$

$$\begin{aligned}
P &\longmapsto P_0\{^{(\mu X,\mathtt{c}_1=1)P_{\mathtt{c}_1}}/_X\}\\
&\longmapsto s[1,\emptyset]\infty^{\mathtt{c}_2=1}_0[(x).P_{0,\mathtt{c}_2}(x)]P_{0,0}(0);(y).(\mu X,\mathtt{c}_1=1)P_{\mathtt{c}_1}\\
&\longmapsto^* s[1,\emptyset]\infty^{\mathtt{c}_2=1}_0[(x).P_{0,\mathtt{c}_2}(x)]\mathtt{call}\langle 0,2+1\rangle;(y).(\mu X,\mathtt{c}_1=1)P_{\mathtt{c}_1}\\
&\longmapsto s[1,\emptyset]\infty^{\mathtt{c}_2=2}_0[(x).P_{0,\mathtt{c}_2}(x)]P_{0,1}(3);(y).(\mu X,\mathtt{c}_1=1)P_{\mathtt{c}_1}\\
&\longmapsto^* s[1,\emptyset]\infty^{\mathtt{c}_2=2}_0[(x).P_{0,\mathtt{c}_2}(x)]\mathtt{exit}\langle 0,5+1\rangle;(y).(\mu X,\mathtt{c}_1=1)P_{\mathtt{c}_1}\\
&\longmapsto (\mu X,\mathtt{c}_1=1)P_{\mathtt{c}_1}\{^6/_y\}\longmapsto P_1\{^{(\mu X,\mathtt{c}_1=2)P_{\mathtt{c}_1}}/_X\}
\end{aligned}$$

4 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 1 (Type Environments)**.** The *global*, *loop* and *session environments* are given by

$$\begin{aligned}
\Gamma &::= \emptyset \mid \Gamma\cdot x{:}\mathrm{S} \mid \Gamma\cdot a{:}G \mid \Gamma\cdot l{:}\mathrm{S}\\
\Theta &::= \emptyset \mid \Theta\cdot X{:}s[\mathsf{r}]t \mid e{:}s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle\\
\Delta &::= \emptyset \mid \Delta\cdot s[\mathsf{r}]{:}T \mid \Delta\cdot s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}
\end{aligned}$$

Global environments with assignments x:S of values to sorts, a:G of shared channels a to global types (for session initialisation), and l:S of labels to sorts as well as session environments with assignments $s[\mathsf{r}]$:T of actors to local types and $s_{\mathsf{r}_1\to\mathsf{r}_2}$:MT of message queues to a list of message types are inherited from [25]. We move assignments X:$s[\mathsf{r}]t$ of process variables to actors and type variables (to check standard recursion) to the new loop environments that also contains assignments e:$s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle$ of loop identifiers to actors and sorts (for the values used to call and exit a loop). Loop environments are used to list active recursion and loops inside their respective bodies.

$$(\mathsf{Req})\ \frac{a{:}G \in \Gamma \quad |\mathrm{R}(G)| = \mathsf{n} \quad \Gamma,\Theta \vdash P \triangleright \Delta \cdot s[\mathsf{n}]{:}G\upharpoonright_{\mathsf{n}}}{\Gamma,\Theta \vdash \overline{a}[\mathsf{n}](s).P \triangleright \Delta} \qquad (\mathsf{If})\ \frac{\Gamma \Vdash e{:}\mathbb{B} \quad \Gamma,\Theta \vdash P \triangleright \Delta \quad \Gamma,\Theta \vdash P' \triangleright \Delta}{\Gamma,\Theta \vdash \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' \triangleright \Delta}$$

$$(\mathsf{Acc})\ \frac{a{:}G \in \Gamma \quad 0 < \mathsf{r} < |\mathrm{R}(G)| \quad \Gamma,\Theta \vdash P \triangleright \Delta \cdot s[\mathsf{r}]{:}G\upharpoonright_{\mathsf{r}}}{\Gamma,\Theta \vdash a[\mathsf{r}](s).P \triangleright \Delta} \qquad (\mathsf{End})\ \frac{\mathsf{noLoop}(\Theta)}{\Gamma,\Theta \vdash \mathbf{0} \triangleright \emptyset}$$

$$(\mathsf{RSend})\ \frac{\Gamma \Vdash y{:}\mathrm{S} \quad \Gamma,\Theta \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\Gamma,\Theta \vdash s[\mathsf{r}_1,\mathsf{r}_2]!_{\mathsf{r}}\langle y\rangle.P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathsf{r}}\langle \mathrm{S}\rangle.T} \qquad (\mathsf{Rec})\ \frac{\Gamma \Vdash n{:}\mathbb{N} \quad \Gamma \cdot \mathsf{c}{:}\mathbb{N},\Theta \cdot X{:}s[\mathsf{r}]t \vdash P \triangleright s[\mathsf{r}]{:}T}{\Gamma,\Theta \vdash (\mu X, \mathsf{c} = n)P \triangleright s[\mathsf{r}]{:}(\mu t, \mathsf{c} = n)T}$$

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

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

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

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

$$(\mathsf{RBran})\ \frac{\forall j \in \mathrm{I}_2.\ \exists i \in \mathrm{I}_1.\ l_i \doteq l'_j \wedge \Gamma,\Theta \vdash P_i \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T_j}{\Gamma,\Theta \vdash s[\mathsf{r}_1,\mathsf{r}_2]?_{\mathsf{r}}\{l_i.P_i\}_{i\in\mathrm{I}_1} \triangleright \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?_{\mathsf{r}}\{l'_i.T_i\}_{i\in\mathrm{I}_2}} \qquad (\mathsf{Res1})\ \frac{x\sharp(\Gamma,\Delta) \quad \Gamma \cdot x{:}\mathrm{S},\Theta \vdash P \triangleright \Delta}{\Gamma,\Theta \vdash (\nu x)P \triangleright \Delta}$$

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

$$(\mathsf{Deleg})\ \frac{\Gamma,\Theta \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T}{\begin{array}{l}\Gamma,\Theta \vdash s[\mathsf{r}_1,\mathsf{r}_2]!\langle\!\langle s'[\mathsf{r}]\rangle\!\rangle.P \triangleright \\ \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!\langle s'[\mathsf{r}]{:}T'\rangle.T \cdot s'[\mathsf{r}]{:}T'\end{array}} \qquad (\mathsf{SRecv})\ \frac{\Gamma,\Theta \vdash P \triangleright \Delta \cdot s[\mathsf{r}_1]{:}T \cdot s'[\mathsf{r}]{:}T'}{\begin{array}{l}\Gamma,\Theta \vdash s[\mathsf{r}_1,\mathsf{r}_2]?((s'[\mathsf{r}])).P \triangleright \\ \Delta \cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]?\langle s'[\mathsf{r}]{:}T'\rangle.T\end{array}}$$

$$(\mathsf{Loop})\ \frac{\begin{array}{c}x,y\sharp(\Gamma,\Delta,s) \quad \mathrm{unr}(T_0) \quad \mathrm{unr}(T_1) \quad \Gamma \cdot x{:}\mathrm{S}_0 \cdot \mathsf{c}{:}\mathbb{N}, e{:}s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle \vdash P_0 \triangleright s[\mathsf{r}]{:}T_0 \\ \Gamma \Vdash n{:}\mathbb{N} \quad \Gamma \cdot \mathsf{c}{:}\mathbb{N}, e{:}s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle \vdash P_1 \triangleright s[\mathsf{r}]{:}T_1 \quad \Gamma \cdot y{:}\mathrm{S}_2,\Theta \vdash P_2 \triangleright \Delta \cdot s[\mathsf{r}]{:}T_2\end{array}}{\Gamma,\Theta \vdash s[\mathsf{r},\mathrm{R}]\infty_e^{\mathsf{c}=n}[(x).P_0]P_1;(y).P_2 \triangleright \Delta \cdot s[\mathsf{r}]{:}[\mathrm{R}]\infty_e^{\mathsf{c}=n}[\langle \mathrm{S}_0\rangle.T_0]T_1;\langle \mathrm{S}_2\rangle.T_2}$$

$$(\mathsf{Call})\ \frac{\Gamma \Vdash e_v{:}\mathrm{S}_0}{\Gamma,\Theta \cdot e{:}s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle \vdash \mathtt{call}\langle e,e_v\rangle \triangleright s[\mathsf{r}]{:}\mathtt{call}\langle e\rangle} \qquad (\mathsf{Exit})\ \frac{\Gamma \Vdash e_v{:}\mathrm{S}_2}{\Gamma,\Theta \cdot e{:}s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle \vdash \mathtt{exit}\langle e,e_v\rangle \triangleright s[\mathsf{r}]{:}T_1}$$

Figure 3: Typing Rules for Fault-Tolerant Systems with Global Escape Loops.

We write $x\sharp\Gamma$, $x\sharp\Theta$, and $x\sharp\Delta$ if x does not occur in Γ, Θ, 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 $l\sharp\Gamma$, $\Theta \cdot X{:}s[\mathsf{r}]t$ implies $X,t\sharp\Theta$, $\Theta \cdot e{:}s[\mathsf{r}]\langle \mathrm{S}_0,\mathrm{S}_2\rangle$ implies $e\sharp\Theta$, $\Delta \cdot s[\mathsf{r}]{:}T$ implies $(\nexists T'.\ s[\mathsf{r}]{:}T' \in \Delta)$, and $\Delta \cdot s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}$ implies $(\nexists \mathrm{MT}'.\ s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}' \in \Delta)$. We naturally extend this operator towards sets, i.e., $\Gamma \cdot \Gamma'$ implies $(\forall A \in \Gamma'.\ \Gamma \cdot A)$, $\Theta \cdot \Theta'$ implies $(\forall A \in \Theta'.\ \Theta \cdot A)$, 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 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 $\Gamma,\Theta \vdash P \triangleright \Delta$, where Γ is a global environment, Θ is a loop environment, $P \in \mathscr{P}$ is a process, and Δ is a session environment. A process P is *well-typed* w.r.t. Γ and Δ if $\Gamma \vdash P \triangleright \Delta$ can be derived from the rules in the Figures 3 and 4. We write $\mathrm{nsr}(\Delta)$ (or $\mathrm{unr}(\Delta)$) if for all types T in Δ we have $\mathrm{nsr}(T)$ (or $\mathrm{unr}(T)$) and if Δ does not contain message queues. With $\Gamma \Vdash y{:}\mathrm{S}$ 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 $x{:}\mathrm{S}_x \in \Gamma$.

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

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

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

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

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

$$(\text{MQExit})\ \frac{\mathrm{eval}(e)=id \quad \Gamma\Vdash v{:}\mathrm{S} \quad \Gamma,\Theta\vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{M}\triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}}{\Gamma,\Theta\vdash s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathtt{exit}\langle id,v\rangle\#\mathrm{M}\triangleright s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathtt{exit}\langle e,\mathrm{S}\rangle\#\mathrm{MT}}$$

Figure 4: Runtime Typing Rules for Fault-Tolerant Systems.

For the rules in Figure 3 we adapted the rules of [25] and extended them by rules for loops. We added the loop environment to all rules that is only relevant for typing recursion and loops. In (End) we add the condition $\mathsf{noLoop}(\Theta)$ that checks that Θ does not contain loop identifiers, to ensure that no branch of a loop program or loop body terminates with $\mathbf{0}$.

(Loop) requires the types of a loop program T_0 and a loop body T_1 to be unreliable ($\mathrm{unr}(T_0)$ and $\mathrm{unr}(T_1)$). It checks the loop program P_0 and the loop body P_1 against their types, but reduces in this check the loop environment to the information for the current loop. This ensures that P_0 and P_1 do not contain free process variables and no calls or exists of surrounding loops. We do not forbid complete recursions or nested loops inside a loop program/body, where the type system ensures their completion before the end of the loop program/body. As in recursion via (Rec), we also reduce the session environment to the actor that initiates this loop. Finally, (Loop) checks the loop continuation P_2 against its type T_2, where Θ and Δ are not reduced. Note that to apply this rule, the expression e used to create the identifier of the loop in the process and the type have to match exactly, i.e., are not evaluated.

(Call) is similar to (Var) and checks that the considered recursion or loop is considered active by the loop environment. Additionally it verifies the sort of the transmitted value. Also (Exit) checks the sort of the transmitted value, requires that the current session environment contains only the actor that invoked the considered loop, and that this loop is considered active by the loop environment. Since (Exit) does not implement any requirement on the type T_1, it does intuitively allow to ignore whatever is left of the loop body.

Figure 4 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 $\overset{s}{\mapsto}$ is given in Figure 5 in [3] 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 $\Delta\cdot s[\mathsf{r}_1]{:}[\mathsf{r}_2]!_{\mathrm{r}}\langle \mathrm{S}\rangle.T\cdot s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}\overset{s}{\mapsto}\Delta\cdot s[\mathsf{r}_1]{:}T\cdot s_{\mathsf{r}_1\to\mathsf{r}_2}{:}\mathrm{MT}\#\langle \mathrm{S}\rangle^{\mathrm{r}}$ that emulates (RSend). Let $\overset{s}{\mapsto}\!\!\!\!\Rightarrow$ denote the reflexive and transitive closure of $\overset{s}{\mapsto}$.

(Res2) and the remaining rules of Figure 4 except for (MQExit) are from [25] extended by the loop environment Θ. (MQExit) checks `exit`-messages on a message queue.

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 patterns in the reduction semantics in Figure 2, 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. As in [24, 25], we leave it for future work to determine what kind of conditions on failure patterns or requirements on their interactions are necessary. Here, we extend the conditions given in [25] by a condition for $\mathtt{FP}_{\mathtt{drop}}$.

Condition 1 (Failure Pattern)**.**

1. *If* $\mathtt{FP}_{\mathtt{crash}}(P,\ldots)$ *then* $\mathrm{nsr}(P)$.
2. *The failure pattern* $\mathtt{FP}_{\mathtt{uget}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots)$ *is always valid.*
3. *The pattern* $\mathtt{FP}_{\mathtt{ml}}(s,\mathsf{r}_1,\mathsf{r}_2,l,\ldots)$ *is valid iff* $\mathtt{FP}_{\mathtt{uskip}}(s,\mathsf{r}_2,\mathsf{r}_1,l,\ldots)$ *is valid.*
4. *If* $\mathtt{FP}_{\mathtt{crash}}(P,\ldots)$ *and* $s[\mathsf{r}] \in \mathrm{A}(P)$ *is an actor then eventually the pattern* $\mathtt{FP}_{\mathtt{uskip}}(s,\mathsf{r}_2,\mathsf{r},l,\ldots)$ *and* $\mathtt{FP}_{\mathtt{wskip}}(s,\mathsf{r}_2,\mathsf{r},l,\ldots)$ *hold for all* r_2,l.
5. *If* $\mathtt{FP}_{\mathtt{crash}}(P,\ldots)$ *and* $s[\mathsf{r}] \in \mathrm{A}(P)$ *then eventually* $\mathtt{FP}_{\mathtt{ml}}(s,\mathsf{r}_1,\mathsf{r},l,\ldots)$ *for all* r_1,l *and* $\mathtt{FP}_{\mathtt{drop}}(\mathsf{r},id)$.
6. *If* $\mathtt{FP}_{\mathtt{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2,\ldots)$ *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.*
7. *If* $s[\mathsf{r}]$ *terminated the loop id then eventually* $\mathtt{FP}_{\mathtt{ml}}(s,\mathsf{r}_1,\mathsf{r},l,\ldots)$ *for all* r_1,l *and* $\mathtt{FP}_{\mathtt{drop}}(\mathsf{r},id)$.
8. *If* $\mathtt{FP}_{\mathtt{drop}}(\mathsf{r},id)$ *then* r *terminated the loop id.*

The crash of a process should not block strongly reliable actions, i.e., only processes with $\mathrm{nsr}(P)$ can crash (Condition 1.1). Condition 1.2 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 1.3 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 1.4) and all messages of a queue towards a crashed receiver can be dropped (Condition 1.5). A weakly reliable branching request 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 $\mathtt{FP}_{\mathtt{wskip}}(s,\mathsf{r}_1,\mathsf{r}_2,\ldots)$ is false as long as $s[\mathsf{r}_2]$ is alive or messages on the respective queue are still in transit (Condition 1.6). The Conditions 1.7 and 1.8 ensure that `exit`-messages can be dropped after the corresponding loop was terminated but not before. Moreover, Condition 1.7 allows to drop messages towards actors of a terminated loop body. Note that such an actor may also be used in the continuation after the loop. By adding *id* to unreliable messages sent from the loop *id*, we could more precisely allow to drop only messages that are intended for the loop. However, the above conditions are sufficient.

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 may not be realised at all but rather observed (which then does not allow to verify them). Because of that, it is an established method to verify the correctness of algorithms w.r.t. given system requirements (e.g. in [11, 21, 27]), even if these system requirements are not verified and often do not hold in all (but only nearly all) cases.

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. For subject reduction we consider only types that were generated from a set of global types, one for each session, using coherence.

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 2 (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:

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

We use $\overset{s}{\mapsto\!\!\!\Rightarrow}$ in the above definition to define coherence for systems that already performed some steps.

Theorem 2 (Subject Reduction)**.** *If Γ, Δ are coherent, $\Gamma, \Theta \vdash P \triangleright \Delta$, and $P \longmapsto P'$, then there is some Δ' such that $\Gamma, \Theta \vdash P' \triangleright \Delta'$.*

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 1.1 to ensure that the type environment of a crashed process cannot contain the types of reliable communication prefixes.

Progress states that no part of a well-typed and coherent system can block other parts, that eventually all matching communication partners 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 [4, 19]. 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 [4] 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, loops are treated in the same way as standard recursion, and exit messages are again ignored, i.e., well-typed for arbitrary types. Remember that loop programs and bodies can act only via the single actor of the loop. 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 often 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.

Theorem 3 (Progress/Session Fidelity)**.** *Let Γ, Δ be coherent, $\Gamma, \Theta \vdash P \triangleright \Delta$, and let P be free of cyclic dependencies between sessions. Assume that in the derivation of $\Gamma, \Theta \vdash P \triangleright \Delta$, 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 or loops, then there exists P' such that $P \longmapsto^* P'$ and P' does not contain any action prefixes.*

The proof of progress relies on the Conditions 1.2–1.8 to ensure that failures cannot block the system: in the failure-free case unreliable messages are eventually received (1.2), the receiver of a lost message can skip (1.3), no receiver is blocked by a crashed sender (1.4), messages towards receivers that crashed or skipped can be dropped (1.5 + 1.3), branching requests cannot be ignored (1.6), and `exit`-messages can be dropped eventually if and only if the corresponding loop was already terminated (1.7 + 1.8).

5 The Rotating Coordinator Algorithm

To illustrate the benefits of our global escape loops, we present an implementation of the rotating coordinator algorithm [11, 16], which is superior to the version without loops presented in [24, 25].

The rotating coordinator algorithm is a small but not trivial consensus algorithm. It was designed for systems with crash failures, but the majority of the algorithm can be implemented with unreliable communication. 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. Then, if at least half of the messages were ack, it sends a weakly reliable global escape containing the decision.

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.

We use the labels p_i and (p_i, r), where $i \in \{1,2,3\}$ specifies the number of the current phase and r is a natural number that specifies the current round. We use p_i as static information and r as runtime information in the labels. Therefore, $p_i \doteq (p_i, \mathsf{r}) \doteq (p_i, \mathsf{r}')$ holds for all i, r, and r'. The additional runtime information can be used in the failure patterns, e.g. to drop outdated messages. 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. Let $\left(\bigodot_{1\le i\le n} \pi_i\right).G$ abbreviate $\pi_1.\ldots.\pi_n.G$ to simplify the presentation, where 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 $\mathsf{r}_1 \to_{\mathrm{u}} \mathsf{r}_2{:}l\langle \mathrm{S}\rangle$ or $\mathsf{r} \to_{\mathrm{w}} \mathsf{R}{:}l_1.T_1 \oplus \ldots \oplus l_n.T_n \oplus l_{\mathrm{d}}$, where the latter case represents a weakly reliable branching prefix (as used in [25]) 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.

$$
\begin{aligned}
G_{\mathrm{rc}}(\mathsf{n}) &\triangleq [\{1,\ldots,\mathsf{n}\}]^{\mathsf{r}}_1\langle \mathrm{S}_{\mathrm{belief}}\rangle.G_0(\mathsf{n},\mathrm{crd}(\mathsf{n},\mathsf{r}))\,;\langle \mathrm{S}_{\mathrm{belief}}\rangle.\mathtt{end}\\
G_0(\mathsf{n},\mathsf{c}) &\triangleq \Big(\bigodot_{1\le \mathsf{i}\le \mathsf{n},\mathsf{i}\neq\mathsf{c}} \mathsf{i} \to_{\mathrm{u}} \mathsf{c}{:}p_1\langle \mathrm{S}_{\mathrm{belief}}\rangle\Big).\Big(\bigodot_{1\le \mathsf{i}\le \mathsf{n},\mathsf{i}\neq\mathsf{c}} \mathsf{c} \to_{\mathrm{u}} \mathsf{i}{:}p_2\langle \mathrm{S}_{\mathrm{belief}}\rangle\Big).\\
&\quad \Big(\bigodot_{1\le \mathsf{i}\le \mathsf{n},\mathsf{i}\neq\mathsf{c}} \mathsf{i} \to_{\mathrm{u}} \mathsf{c}{:}p_3\langle \mathrm{S}_{\mathrm{ack}}\rangle\Big).\mathtt{call}\langle 1\rangle
\end{aligned}
$$

$G_{\mathrm{rc}}(\mathsf{n})$ specifies a loop with identifier 1 and counter r, where r is used as the round number. The coordinator c of round r is calculated by $\mathrm{crd}(\mathsf{n},\mathsf{r}) \triangleq (\mathsf{r} \bmod \mathsf{n}) + 1$. Then, $G_0(\mathsf{n},\mathsf{c})$ specifies the loop program that implements one round of the algorithm. The three $\odot$ specify the Phases 1–3 of the algorithm within a single round. Phase 4 is only specified by $\mathtt{call}\langle 1\rangle$, since there is no `exit` type.

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 can terminate the protocol

by sending a global escape message containing the decision. All interactions in the specification are unreliable.

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

$$\begin{aligned}
Sys\left(\mathsf{n},\vec{V}\right) &\triangleq \overline{a}[\mathsf{n}](s).P_{\mathrm{rc}}(\mathsf{n},\mathsf{n},v_{\mathsf{n}}) \mid \prod_{1\leq \mathsf{i}<\mathsf{n}} a[\mathsf{i}](s).P_{\mathrm{rc}}(\mathsf{i},\mathsf{n},v_{\mathsf{i}})\\
P_{\mathrm{rc}}(\mathsf{i},\mathsf{n},v_{\mathsf{i}}) &\triangleq s[\mathsf{i},\{1,\ldots,\mathsf{n}\}\setminus\{\mathsf{i}\}]\infty_1^{\mathsf{r}=0}[(v_{\mathsf{i}}).P_0(\mathsf{i},\mathsf{n},\mathsf{r},v_{\mathsf{i}})]\mathtt{call}\langle 1,v_{\mathsf{i}}\rangle;(v).\mathbf{0}\\
P_0(\mathsf{i},\mathsf{n},\mathsf{r},v_{\mathsf{i}}) &\triangleq \mathtt{if}\ \mathsf{i}=\mathrm{crd}(\mathsf{n},\mathsf{r})\ \mathtt{then}\ P_1^{\mathrm{C}}(\mathsf{i},\mathsf{n},\mathsf{r},v_{\mathsf{i}})\ \mathtt{else}\ P_1^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathrm{crd}(\mathsf{n},\mathsf{r}),\mathsf{r},v_{\mathsf{i}})
\end{aligned}$$

$Sys\left(\mathsf{n},\vec{V}\right)$ describes the session initialisation of a system with n participants and the (initial) knowledge $\vec{V}=\{v_{\mathsf{i}}\mid 1\leq\mathsf{i}\leq\mathsf{n}\}$, where v_{i} is the initial belief of role i. Let $\left|\vec{V}\right| \triangleq |\{\mathsf{i}\mid v_{\mathsf{i}}\neq\bot\}|$ return the number of non-empty entries. $P_{\mathrm{rc}}(\mathsf{i},\mathsf{n},v_{\mathsf{i}})$ describes a process i in a set of n processes. Each process is described as a loop with identifier 1 and counter r, where the loop program executes the round r of the algorithm. Once a decision is reached and the loop ends, the loop continuation is instantiated with the decision value.

$$\begin{aligned}
P_1^{\mathrm{C}}(\mathsf{c},\mathsf{n},\mathsf{r},v_{\mathsf{c}}) &\triangleq \Big(\bigodot_{1\leq\mathsf{i}\leq\mathsf{n},\mathsf{i}\neq\mathsf{c}} s[\mathsf{c},\mathsf{i}]?_{\mathrm{u}}(p_1,\mathsf{r})\langle\bot\rangle(v_{\mathsf{i}})\Big).\\
&\qquad \mathtt{if}\ \left|\vec{V}\right|\geq\left\lceil\frac{\mathsf{n}-1}{2}\right\rceil\ \mathtt{then}\ P_2^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\mathsf{r},\mathrm{best}(\vec{V}),\mathrm{best}(\vec{V})\right)\ \mathtt{else}\ P_2^{\mathrm{C}}(\mathsf{c},\mathsf{n},\mathsf{r},v_{\mathsf{c}},\bot)\\
P_1^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},v_{\mathsf{i}}) &\triangleq s[\mathsf{i},\mathsf{c}]!_{\mathrm{u}}(p_1,\mathsf{r})\langle v_{\mathsf{i}}\rangle.P_2^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},v_{\mathsf{i}})
\end{aligned}$$

In Phase 1, every non-coordinator $P_1^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},v_{\mathsf{i}})$ sends its own belief via unreliable communication to the coordinator and proceeds to Phase 2. The coordinator receives (some of) these messages and writes each one into its knowledge vector before proceeding 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 returns the best belief value. Otherwise, it continues to use its own belief. 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.

$$\begin{aligned}
P_2^{\mathrm{C}}(\mathsf{c},\mathsf{n},\mathsf{r},v_{\mathsf{c}},x) &\triangleq \Big(\bigodot_{1\leq\mathsf{i}\leq\mathsf{n},\mathsf{i}\neq\mathsf{c}} s[\mathsf{c},\mathsf{i}]!_{\mathrm{u}}(p_2,\mathsf{r})\langle x\rangle\Big).P_3^{\mathrm{C}}(\mathsf{c},\mathsf{n},\mathsf{r},v_{\mathsf{c}})\\
P_2^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},v_{\mathsf{i}}) &\triangleq s[\mathsf{i},\mathsf{c}]?_{\mathrm{u}}(p_2,\mathsf{r})\langle\bot\rangle(x).\\
&\qquad \mathtt{if}\ x=\bot\ \mathtt{then}\ P_3^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},v_{\mathsf{i}},\mathtt{f})\ \mathtt{else}\ P_3^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},x,\mathtt{t})
\end{aligned}$$

In Phase 2, the coordinator sends its updated belief to all other processes via unreliable communication and proceeds. Note that x is either $\bot$ or the best belief identified in Phase 1. If a non-coordinator process successfully receives a belief other than $\bot$, 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 $\bot$ was received, the process proceeds to Phase 3 with the Boolean value $\mathtt{f}$, signalling nack.

$$\begin{aligned}
P_3^{\mathrm{C}}(\mathsf{c},\mathsf{n},\mathsf{r},v_{\mathsf{c}}) &\triangleq \Big(\bigodot_{1\leq\mathsf{i}\leq\mathsf{n},\mathsf{i}\neq\mathsf{c}} s[\mathsf{c},\mathsf{i}]?_{\mathrm{u}}(p_3,\mathsf{r})\langle\bot\rangle(v_{\mathsf{i}})\Big).P_4^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right)\\
P_3^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},\mathsf{r},v_{\mathsf{i}},b) &\triangleq s[\mathsf{i},\mathsf{c}]!_{\mathrm{u}}(p_3,\mathsf{r})\langle b\rangle.P_4^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},v_{\mathsf{i}})
\end{aligned}$$

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.

$$P_4^{\mathrm{C}}\left(\mathsf{c},\mathsf{n},\vec{V}\right) \triangleq \texttt{if } \mathrm{ack}(\vec{V}) \geq \left\lceil \frac{\mathsf{n}-1}{2} \right\rceil \texttt{ then exit}\langle 1, v_{\mathsf{c}}\rangle \texttt{ else call}\langle 1, v_{\mathsf{c}}\rangle$$
$$P_4^{\mathrm{NC}}(\mathsf{i},\mathsf{n},\mathsf{c},v_{\mathsf{i}}) \triangleq \texttt{call}\langle 1, v_{\mathsf{i}}\rangle$$

In Phase 4, all non-coordinators move on to the next round. The coordinator checks if at least half of the non-coordinator roles signalled acknowledgement, utilising the function ack() to count. If it received enough acknowledgments, it sends a global escape message containing the decision value, which causes all participants to eventually terminate. Otherwise, the coordinator continues with the next round.

The main difference between this implementation and the previous version without loops [25] lies in Phase 4. In the previous version, the coordinator transmitted the decision via broadcasting one of the labels *Zero*, *One*, or l_{d}. The first two labels represented a decision and terminated the protocol, whereas the default label l_{d} specified the need for another round:

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

where $\mathscr{I} = \{1,\ldots,\mathsf{n}\} \setminus \{\mathsf{c}\}$ and the missing continuation after l_{d} is implemented by the next round. This caused all non-coordinators to wait for the coordinator's decision before proceeding to the next round.

In our new implementation, presented above, non-coordinators can proceed to the next round immediately after Phase 3. They can also skip entire rounds by suspecting the coordinator. Thus, processes can diverge as freely in their rounds as in the original rotating coordinator algorithm [11]. Exiting the loop mimics the so-called *reliable broadcast* of the original algorithm.

Chandra and Toueg [11] introduce the failure detector $\Diamond\mathscr{S}$ that 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 $\mathtt{FP}_{\mathtt{uskip}}$ 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 $\mathtt{FP}_{\mathtt{uskip}}$ may allow to suspect processes that are not crashed after the coordinator received enough messages. In all other cases, this pattern eventually returns true iff the respective sender is crashed. Moreover, $\mathtt{FP}_{\mathtt{uskip}}$ is true for outdated messages, i.e., messages with a round number smaller than the current round of the process.

$\mathtt{FP}_{\mathtt{uget}}$ returns true. To prevent the system from becoming blocked, $\mathtt{FP}_{\mathtt{ml}}$ and $\mathtt{FP}_{\mathtt{drop}}$ eventually return true for messages that cannot be consumed, i.e., for messages with label p_2 that were suspected using $\Diamond\mathscr{S}$, skipped p_1/p_3-messages, messages from old rounds, and messages after the termination of the loop. Otherwise, $\mathtt{FP}_{\mathtt{ml}}$ and $\mathtt{FP}_{\mathtt{drop}}$ returns false. By the system requirements in [11], 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 $\mathtt{FP}_{\mathtt{crash}}$ by false if only half of the processes are alive and true otherwise. These failure patterns satisfy the Conditions 1.1–1.8.

The proof of termination, agreement, and validity of the algorithm is discussed in [25]. The main difference is that there may be multiple `exit`-messages, but the requirement on the majority in Phase 4 ensures that they all carry the same decision value.

6 Conclusions

We present an unreliable loop construct with weakly reliable global escape for fault-tolerant multiparty session types (FTMPST) for systems that may suffer from message loss or crash failures. We prove subject reduction and progress and present a small but relevant case study.

Currently we require all actions within loop programs/bodies to be unreliable. This ensures that a communication partner is not blocked if a loop is terminated. An interesting question for further work is how to relax this requirement. For instance, we may allow for a variant of weakly reliable branching within loop programs/bodies, where moving to the default branch is not only allowed if the sender is suspected to be crashed but also if the receiver suspects that the sender already terminated its loop or at least already moved to another loop iteration.

Moreover, there are a couple of open problems from [24, 25]. A really difficult challenge is to extend branching to at least some kind of message loss, while maintaining the strong properties of the type system and ensuring that no two alive processes move to different branches.

We also 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. Unreliable variants of session initialisation open a new perspective on MPST-frameworks such as [12] with dynamically changing network topologies and sessions for that the number of roles is determined at run-time.

As in [25] we fix one set of conditions on failure patterns to prove subject reduction and progress. We can also think of other sets of conditions. As already mentioned, we can improve Condition 1.7 by explicitly using the *id* of loops in unreliable messages. We can also use the failure pattern $\mathtt{FP}_{\mathtt{uget}}$ to reject the reception of outdated messages. Therefore, we drop Condition 1.2 and instead require for each message m whose reception is refused that $\mathtt{FP}_{\mathtt{ml}}$ ensures that m is eventually dropped from the respective queue and that $\mathtt{FP}_{\mathtt{uskip}}$ 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 5, namely for validity, agreement, and termination. The approach in [26] 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

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

[2] Marcos Kawazoe Aguilera, Wei Chen & Sam Toueg (1997): *Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication*. In: *Proc. of WDAG, LNCS* 1320, Springer, pp. 126–140, doi:10.1007/BFb0030680.

[3] Lukas Bartl, Julian Linne & Kirstin Peters (2025): *Fault-Tolerant Multiparty Session Types with Global Escape Loops (Technical Report)*. Technical Report. Submitted to `https://hal.science`.

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

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

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

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

[8] Marco Carbone, Kohei Honda & Nobuko Yoshida (2008): *Structured Interactional Exceptions in Session Types*. In: *Proc. of CONCUR*, *LNCS* 5201, Springer, pp. 402–417, doi:10.1007/978-3-540-85361-9_32.

[9] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2017): *Concurrent Reversible Sessions*. In: *Proc. of CONCUR*, *LIPIcs* 85, pp. 30:1–30:17, doi:10.4230/LIPIcs.CONCUR.2017.30.

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

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

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

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

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

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

[16] Rachele Fuzzati, Massimo Merro & Uwe Nestmann (2007): *Distributed Consensus, revisited*. *Acta Informatica*, pp. 377–425, doi:10.1007/s00236-007-0052-1.

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

[18] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty Asynchronous Session Types*. In: *Proc. of POPL*, 43, ACM, pp. 273–284, doi:10.1145/1328438.1328472.

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

[20] Dimitrios Kouzapas, Ramūnas Gutkovas & Simon J. Gay (2014): *Session Types for Broadcasting*. In: *Proc. of PLACES*, *EPTCS* 155, pp. 25–31, doi:10.4204/EPTCS.155.4.

[21] Leslie Lamport (2001): *Paxos Made Simple*. *ACM Sigact News* 32(4), pp. 51–58. Available at https://www.microsoft.com/en-us/research/publication/paxos-made-simple.

[22] Nancy A. Lynch (1996): *Distributed Algorithms*. Morgan Kaufmann. Available at https://dl.acm.org/doi/book/10.5555/2821576.

[23] Rumyana Neykova & Nobuko Yoshida (2017): *Let it recover: multiparty protocol-induced recovery*. In: *Proc. of CC*, ACM, pp. 98–108, doi:10.1145/3033019.3033031.

[24] Kirstin Peters, Uwe Nestmann & Christoph Wagner (2022): *Fault-Tolerant Multiparty Session Types*. In: *Proc. of FORTE*, *LNCS* 13273, Springer, pp. 93–113, doi:10.1007/978-3-031-08679-3_7.

[25] Kirstin Peters, Uwe Nestmann & Christoph Wagner (2023): *FTMPST: Fault-Tolerant Multiparty Session Types*. *Logical Methods in Computer Science* 19(4), doi:10.46298/LMCS-19(4:14)2023.

[26] Kirstin Peters, Christoph Wagner & Uwe Nestmann (2019): *Taming Concurrency for Verification Using Multiparty Session Types*. In: *Proc. of ICTAC*, *LNCS* 11884, pp. 196–215, doi:10.1007/978-3-030-32505-3_12.

[27] Maarten van Steen & Andrew S. Tanenbaum (2017): *Distributed Systems*, 3rd edition. Maarten van Steen. Available at `https://www.distributed-systems.net/index.php/books/ds3`.

[28] Gerard Tel (2000): *Introduction to Distributed Algorithms*, 2nd edition. Cambridge University Press, doi:10.1017/CBO9781139168724.

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