

# Generalised Multiparty Session Types with Crash-Stop Failures

Adam D. Barwell  



Imperial College London

Alceste Scalas  

DTU Compute — Technical University of Denmark

Nobuko Yoshida  

Imperial College London

Fangyi Zhou  

Imperial College London

---

## Abstract

---

Session types enable the specification and verification of communicating systems. However, their theory often assumes that processes never fail. To address this limitation, we present a generalised multiparty session type (MPST) theory with *crash-stop failures*, where processes can crash arbitrarily.

Our new theory validates more protocols and processes w.r.t. previous work. We apply minimal syntactic changes to standard session  $\pi$ -calculus and types: we model crashes and their handling semantically, with a generalised MPST typing system parametric on a behavioural safety property. We cover the spectrum between fully reliable and fully unreliable sessions, via *optional reliability assumptions*, and prove type safety and protocol conformance in the presence of crash-stop failures.

Introducing crash-stop failures has non-trivial consequences: writing correct processes that handle all crash scenarios can be difficult. Yet, our generalised MPST theory allows us to tame this complexity, via model checking, to validate whether a multiparty session satisfies desired behavioural properties, e.g. deadlock-freedom or liveness, even in presence of crashes. We implement our approach using the mCRL2 model checker, and evaluate it with examples extended from the literature.

**2012 ACM Subject Classification** Theory of computation → Distributed computing models; Theory of computation → Process calculi; Software and its engineering → Model checking

**Keywords and phrases** Session Types, Concurrency, Failure Handling, Model Checking

**Digital Object Identifier** 10.4230/LIPIcs.CONCUR.2022.3

**Related Version** Extended technical report: <https://doi.org/10.48550/arXiv.2207.02015>

**Funding** Work supported by: EU Horizon 2020 project 830929; EPSRC grants EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1, EP/V000462/1, and NCSS/EPSRC VeTSS; Danmarks Industriens Fond 2020-0489.

## 1 Introduction

Multiparty session types (MPST) [20] provide a typing discipline for message-passing processes. The theory ensures well-typed processes enjoy desirable properties, a.k.a. *the Session Theorems*: type safety (processes communicate without errors), protocol conformance (a.k.a. *session fidelity*, processes behave according to their types), deadlock-freedom (processes do not get stuck), and liveness (input/output actions eventually succeed). Researchers devote significant effort into integrating session types in programming languages and tools [16].

A common assumption in session type theory is that everything is reliable and there are no failures, which is often unrealistic in real-world systems. So, we pose a question: how can we better model systems *with failures*, and make session types less idealistic?



© Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou;  
licensed under Creative Commons License CC-BY 4.0

33rd International Conference on Concurrency Theory (CONCUR 2022).

Editors: Bartek Klin, Slawomir Lasota, and Anca Muscholl; Article No. 3; pp. 3:1–3:25



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we take steps towards bridging the gap between theory and practice with a new *generalised* multiparty session type theory that models failures with *crash-stop* semantics [5, §2.2]: processes may crash, and crashed processes stop interacting with the world. This model is standard in distributed systems, and is used in related work on session types with error-handling capabilities [33, 34]. However, unlike previous work, we allow *any* process to crash arbitrarily, and support optional assumptions on non-crashing processes.

In our new theory, we add crashing and crash handling semantics to processes and session types. With minimal changes to the standard surface syntax, we model a variety of subtle, complex behaviours arising from unreliable communicating processes. An active process  $P$  may crash arbitrarily, and a process  $Q$  interacting with  $P$  might need to be prepared to handle possible crashes. Messages sent from  $Q$  to a crashed  $P$  are lost – but if  $Q$  tries to receive from  $P$ , then  $Q$  can detect that  $P$  has crashed, and take a crash handling branch. Meanwhile, another process  $R$  may (or may not) have detected  $P$ 's crash, and may be handling it – and in either case, any interaction between  $Q$  and  $R$  should remain correct.

Our MPST theory is generalised in two aspects: (1) we introduce *optional reliability assumptions*, so we can model a mixture of reliable and unreliable communicating peers; and (2) our type system is parametric on a type-level behavioural property  $\varphi$  which can be instantiated as safety, deadlock freedom, liveness, *etc.* (in the style of [29]), while accounting for potential crashes. We prove *session fidelity*, showing how type-level properties transfer to well-typed processes; we also prove that our new theory satisfies other Session Theorems of MPST, while (unlike previous work) being resilient to arbitrary crash-stop failures.

With *optional reliability assumptions*, one may declare that some peers will never crash for the duration of the protocol. Such optional assumptions allow for simplifying protocols and programs: if a peer is assumed reliable, the other peers can interact with it without needing to handle its crashes. By making such assumptions explicit and customisable, our theory supports a *spectrum* of scenarios ranging from only having sessions with reliable peers (thus subsuming classic MPST works [20, 29]), to having no reliable peers at all.

As in the real world, a system with crash-stop failures can have subtle complex behaviours; hence, writing protocols and processes where all possible crash scenarios are correctly handled can be hard. This highlights a further benefit of our generalised theory: we formalise our behavioural properties as modal  $\mu$ -calculus formulæ, and verify them with a model checker. To show the feasibility of our approach, we present an accompanying tool, utilising the mCRL2 model checker [4], for verifying session properties under optional reliability assumptions.

**Overview.** Session typing systems assign *session types* (a.k.a. local types) to communication channels, used by processes to send and receive messages. In essence, a session type describes a *protocol*: how a *role* is expected to interact with other roles in a multiparty session. The type system checks whether a process implements desired protocols.

As an example, consider a simple Domain Name System (DNS) scenario: a client  $p$  queries a server  $q$  for an IP address of a host name. With classic session types (without crashes), we use the type  $T_p = q \oplus \text{req}.q \& \text{res}$  to represent the client  $p$ 's behaviour: first sending ( $\oplus$ ) a **request** message to server  $q$ , and then receiving ( $\&$ ) a **response** from  $q$ . The server implements a dual type  $T_q = p \& \text{req}.p \oplus \text{res}$ , who receives a **request** from client  $p$ , and then sends a **response** to  $p$ . We can write a process  $Q = s[q][p] \& \text{req}.s[q][p] \oplus \text{res}.\mathbf{0}$  for the server. Using  $T_q$ , we type-check the channel (a.k.a. session endpoint)  $s[q]$ , where  $Q$  plays the role  $q$  on session  $s$ . Here,  $Q$  type-checks – it uses channel  $s[q]$  correctly, according to type  $T_q$ .

In this work, we augment the classic session types theory by introducing process failures with *crash-stop* semantics [5, §2.2]. We adopt the following failure model: (1) processes have



■ **Figure 1** Transition systems (based on Def. 7, with labels omitted) generated from the DNS examples. Left: without crashes/handling. Right: with crashes (for  $q$ ) and crash handling.

*crash-stop* failures, i.e. they may crash and do not recover; (2) communication channels deliver messages in order, without losses (unless the recipient has crashed); (3) each process has a failure detector [11], so a process trying to receive from a crashed peer accurately detects the crash. The combination of (1), (2), and (3) is called the *crash-fail* model in [5, §2.6.2].

We now revise our DNS example in the presence of failures. Let us assume that the server  $q$  may crash, whereas the client  $p$  remains reliable. The client  $p$  may now send its **request** to a new *failover server*  $r$  (assumed reliable for simplicity). We represent this scenario by a type for the new failover server  $T'_r$ , and a new branch in  $T'_p$  for handling  $q$ 's crash:

$$T'_p = q \oplus \text{req.}q \& \left\{ \begin{array}{l} \text{res} \\ \text{crash.}r \oplus \text{req.}r \& \text{res} \end{array} \right\} \quad \begin{array}{l} T'_q = p \& \text{req.}p \oplus \text{res} \\ T'_r = q \& \text{crash.}p \& \text{req.}p \oplus \text{res} \end{array}$$

Here,  $T'_p$  states that client  $p$  first sends a message to the unreliable server  $q$ ; then,  $p$  expects a **response** from  $q$ . If  $q$  crashes, the client  $p$  detects the crash and handles it (via the new **crash** handling branch) by **requesting** from the failover server  $r$ . Meanwhile,  $r$  *also* detects whether  $q$  has crashed. If so,  $r$  activates its **crash** handling branch and handles  $p$ 's **request**.

In our model, crash detection and handling is done on the receiving side, e.g.  $T'_p$  detects whether  $q$  has **crashed** when waiting for a **response**, while  $T'_r$  monitors whether  $q$  has crashed. Handling crashes when receiving messages from a reliable role is unnecessary, e.g. the server  $q$  does not need crash handling when it receives from the (reliable) client  $p$ ; similarly, the (reliable) roles  $p$  and  $r$  interact without crash handling. This failure model is reflected in the semantics of both processes and session types in our work. Unlike classic MPST works, we allow processes to crash *arbitrarily* while attempting inputs or outputs (§2). When a process crashes, the channel endpoints held by the process also crash, and are assigned the new type **stop** (§3). E.g. when the server process  $Q$  crashes, the endpoint  $s[q]$  held by  $Q$  becomes a crashed endpoint  $s[q]_{\downarrow}$ ; accordingly, the server type  $T'_q$  advances to **stop** to reflect the crash.

To ensure that communicating processes are type-safe even in the presence of crashes, we require their session types to satisfy a *safety property* accounting for possible crashes (Def. 8), which can be refined, e.g. as deadlock-freedom or liveness (Def. 18). We prove subject reduction, session fidelity, and various process properties (deadlock-freedom, liveness, *etc.*) even in the presence of crashes and optional reliability assumptions (Thms. 11, 15, 20).

Despite minimal changes to the surface syntax of session types and processes, the semantics surrounding crashes introduce subtle behaviours and increase complexity. Taking the DNS examples above, we compare the sizes of their (labelled) transition systems in Fig. 1 (based on Def. 7): the original system (left, two roles  $p$  and  $q$ , no crashes) has 10 states and 15 transitions; and the revised system (right,  $q$  may crash, with a new role  $r$ ) has 101 states and 427 transitions. We discuss another, more complex example in §4. Checking whether a given combination of session types with possible crashes is safe, deadlock-free, or live, can be challenging due to non-trivial behaviours and increased model size arising from crashes and crash handling. To tackle this, we show how to automatically verify such type-level properties

by representing them as modal  $\mu$ -calculus formulæ via the mCRL2 model checker [4].

**Contributions and Structure.** In §2 we introduce a multiparty session  $\pi$ -calculus (with minimal changes to the standard syntax) giving crash and crash handling semantics modelling *crash-stop* failures. In §3 we present *multiparty session types with crashes*: they describe how communication channels should be used to send/receive messages, and handle crashes. We formalise the semantics of collections of local types under *optional* reliability assumptions; we introduce a type system, and prove the Session Theorems: type safety, protocol conformance, and process properties (deadlock-freedom, termination, liveness, *etc.*) in Thms. 11, 15, and 20. In §4 we show how model checking can be incorporated to verify our behavioural properties, by expressing them as modal  $\mu$ -calculus formulæ. We discuss related work and conclude in §5. Appendices §C and §D include additional examples and more details about the tool implementing our theory using the mCRL2 model checker. Further appendices with proofs are available in a separate technical report [3].

## 2 Multiparty Session Calculus with Crash-Stop Semantics

In this section, we formalise the syntax and operational semantics of our multiparty session  $\pi$ -calculus, where a process can fail arbitrarily, and crashes can be detected and handled by receiving processes. For clarity of presentation, we formalise a synchronous semantics.

**Syntax of Processes.** Our multiparty session  $\pi$ -calculus models processes that interact via multiparty channels, and may arbitrarily crash. For simplicity of presentation, our calculus is streamlined to focus on communication; standard extensions, e.g. with expressions and “if...then...else” statements, are routine and orthogonal to our formulation.

► **Definition 1** (Syntax of Multiparty Session  $\pi$ -Calculus). *Let  $\mathbf{p}, \mathbf{q}, \dots$  denote roles belonging to a set  $\mathfrak{R}$ ; let  $s, s', \dots$  denote sessions; let  $x, y, \dots$  denote variables; let  $\mathbf{m}, \mathbf{m}', \dots$  denote message labels; let  $X, Y, \dots$  denote process variables. The multiparty session  $\pi$ -calculus syntax is:*

$$\begin{array}{ll}
c ::= x \mid s[\mathbf{p}] & \text{(variable or channel for session } s \text{ with role } \mathbf{p}) \\
d ::= v \mid c & \text{(basic value, variable, or channel with role)} \\
w ::= v \mid s[\mathbf{p}] & \text{(basic value or channel with role)} \\
P, Q ::= \mathbf{0} \mid (\nu s)P \mid P \mid Q & \text{(inaction, restriction, parallel composition)} \\
& c[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle.P \quad \text{(where } \mathbf{m} \neq \text{crash)} \quad \text{(selection towards role } \mathbf{q}) \\
& c[\mathbf{q}] \& \{\mathbf{m}_i(x_i).P_i\}_{i \in I} & \text{(branching from role } \mathbf{q} \text{ with an index set } I \neq \emptyset) \\
& \text{def } D \text{ in } P \mid X(\tilde{d}) & \text{(process definition, process call)} \\
& \text{err} \mid s[\mathbf{p}] \downarrow & \text{(error, crashed channel endpoint)} \\
D ::= X(\tilde{x}) = P & \text{(declaration of process variable } X)
\end{array}$$

We write  $\Pi_{i \in I} P_i$  for the parallel composition of processes  $P_i$ . Restriction, branching, and process definitions and declarations act as binders, as expected;  $\text{fc}(P)$  is the set of free channels with roles in  $P$  (including  $s[\mathbf{p}]$  in  $s[\mathbf{p}] \downarrow$ ), and  $\text{fv}(P)$  is the set of free variables in  $P$ . Noticeable changes w.r.t. standard session calculi are *highlighted*.

Our calculus (Def. 1) includes basic values  $v$  (e.g. unit  $()$ , integers, strings), channels with roles (a.k.a. session endpoints)  $s[\mathbf{p}]$ , session scope restriction  $(\nu s)P$ , inaction  $\mathbf{0}$ , parallel composition  $P \mid Q$ , process definition  $\text{def } D \text{ in } P$ , process call  $X(\tilde{d})$ , and error **err**. *Selection* (a.k.a. internal choice)  $c[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle.P$  sends a message  $\mathbf{m}$  with payload  $d$  to role  $\mathbf{q}$  via endpoint  $c$ , where  $c$  may be a variable or channel with role, while  $d$  may also be a basic value. *Branching* (a.k.a. external choice)  $c[\mathbf{q}] \& \{\mathbf{m}_i(x_i).P_i\}_{i \in I}$  expects to receive a message  $\mathbf{m}_i$  (for some  $i \in I$ )

$$\begin{array}{l}
\text{[R-}\oplus\&] \quad s[\mathbf{p}][\mathbf{q}]\&\{\mathbf{m}_i(x_i).P_i\}_{i \in I} \mid s[\mathbf{q}][\mathbf{p}]\oplus\mathbf{m}_k\langle w \rangle.Q \rightarrow P_k\{w/x_k\} \mid Q \quad \text{if } k \in I \\
\text{[R-ERR]} \quad s[\mathbf{p}][\mathbf{q}]\&\{\mathbf{m}_i(x_i).P_i\}_{i \in I} \mid s[\mathbf{q}][\mathbf{p}]\oplus\mathbf{m}\langle w \rangle.Q \rightarrow \mathbf{err} \quad \text{if } \forall i \in I : \mathbf{m}_i \neq \mathbf{m} \\
\text{[R-X]} \quad \text{def } X(x_1, \dots, x_n) = P \text{ in } (X\langle w_1, \dots, w_n \rangle \mid Q) \\
\quad \rightarrow \text{def } X(x_1, \dots, x_n) = P \text{ in } (P\{w_1/x_1\} \cdots \{w_n/x_n\} \mid Q) \\
\text{[R-CTX]} \quad P \rightarrow P' \text{ implies } \mathbb{C}[P] \rightarrow \mathbb{C}[P'] \\
\text{[R-}\equiv] \quad P' \equiv P \text{ and } P \rightarrow Q \text{ and } Q \equiv Q' \text{ implies } P' \rightarrow Q' \\
\text{[R-}\zeta\oplus] \quad P = s[\mathbf{p}][\mathbf{q}]\oplus\mathbf{m}\langle w \rangle.P' \rightarrow \prod_{j \in J} s_j[\mathbf{p}_j]\zeta \quad \text{where } \{s_j[\mathbf{p}_j]\}_{j \in J} = \text{fc}(P) \\
\text{[R-}\zeta\&] \quad P = s[\mathbf{p}][\mathbf{q}]\&\{\mathbf{m}_i(x_i).P_i\}_{i \in I} \rightarrow \prod_{j \in J} s_j[\mathbf{p}_j]\zeta \quad \text{where } \{s_j[\mathbf{p}_j]\}_{j \in J} = \text{fc}(P) \\
\text{[R-}\zeta\mathbf{m}B] \quad s[\mathbf{p}]\zeta \mid s[\mathbf{q}][\mathbf{p}]\oplus\mathbf{m}\langle v \rangle.Q' \rightarrow s[\mathbf{p}]\zeta \mid Q' \\
\text{[R-}\zeta\mathbf{m}] \quad s[\mathbf{p}]\zeta \mid s[\mathbf{q}][\mathbf{p}]\oplus\mathbf{m}\langle s'[\mathbf{r}] \rangle.Q' \rightarrow s[\mathbf{p}]\zeta \mid s'[\mathbf{r}]\zeta \mid Q' \\
\text{[R-}\odot] \quad s[\mathbf{p}][\mathbf{q}]\&\{\mathbf{m}_i(x_i).P_i, \mathbf{crash}.P'\}_{i \in I} \mid s[\mathbf{q}]\zeta \rightarrow P' \mid s[\mathbf{q}]\zeta
\end{array}$$

■ **Figure 2** Semantics of our session  $\pi$ -calculus. Rule [R- $\equiv$ ] uses the congruence  $\equiv$  defined in § A.

from role  $\mathbf{q}$  via endpoint  $c$ , and then continues as  $P_i$ . Importantly, a process implements crash detection by “receiving” the special message label  $\mathbf{crash}$  in an external choice; such special message *cannot* be sent by any process (side condition  $\mathbf{m} \neq \mathbf{crash}$  in selection). For example,  $s[\mathbf{p}][\mathbf{q}]\&\{\mathbf{m}(x).P, \mathbf{crash}.P'\}$  is a process that uses the session endpoint  $s[\mathbf{p}]$  to receive message  $\mathbf{m}$  from  $\mathbf{q}$ , but if  $\mathbf{q}$  has crashed, then the process continues as  $P'$ . Finally, our calculus includes *crashed session endpoints*  $s[\mathbf{p}]\zeta$ , denoting that the endpoint for role  $\mathbf{p}$  in session  $s$  has crashed.

**Operational Semantics.** We give the operational semantics of our session  $\pi$ -calculus in Def. 2, using a standard *structural congruence* extended with a new *crash elimination rule* which garbage-collects sessions where all endpoints are crashed: (full congruence rules in § A)

$$(\nu s)(s[\mathbf{p}_1]\zeta \mid \cdots \mid s[\mathbf{p}_n]\zeta) \equiv \mathbf{0} \quad \text{[C-CRASHELIM]}$$

► **Definition 2.** A reduction context  $\mathbb{C}$  is defined as:  $\mathbb{C} ::= \mathbb{C} \mid P \mid (\nu s)\mathbb{C} \mid \text{def } D \text{ in } \mathbb{C} \mid []$ . The reduction  $\rightarrow$  is defined in Fig. 2; we write  $\rightarrow^+ / \rightarrow^*$  for its transitive / reflexive-transitive closure. We write  $P \not\rightarrow$  iff  $\nexists P'$  such that  $P \rightarrow P'$  is derivable without rules [R- $\zeta\oplus$ ] and [R- $\zeta\&$ ] (i.e.  $P$  is stuck, unless a crash occurs). We say  $P$  has an error iff  $\exists \mathbb{C}$  with  $P = \mathbb{C}[\mathbf{err}]$ .

Part of our operational semantics rules in Fig. 2 are standard. Rule [R- $\oplus\&$ ] describes a communication on session  $s$  between receiver  $\mathbf{p}$  and sender  $\mathbf{q}$ , if the sent message  $\mathbf{m}_k$  can be handled by the receiver ( $k \in I$ ); otherwise, a message label mismatch causes an **error** via rule [R-ERR]. Rule [R-X] expands process definitions when called. Rules [R-CTX] and [R- $\equiv$ ] allow processes to reduce under reduction contexts and modulo structural congruence.

The remaining rules in Fig. 2 (**highlighted**) are novel: they model crashes, and crash handling. Rules [R- $\zeta\oplus$ ] and [R- $\zeta\&$ ] state that a process  $P$  may crash while attempting any selection or branching operation, respectively; when  $P$  crashes, it reduces to a parallel composition where all the channel endpoints held by  $P$  are crashed. The *lost message rules* [R- $\zeta\mathbf{m}B$ ] and [R- $\zeta\mathbf{m}$ ] state that if a process sends a message to a crashed endpoint, then the message is lost; if the message payload is a session endpoint  $s'[\mathbf{r}]$ , then it becomes crashed. Finally, the *crash handling rule* [R- $\odot$ ] states that if a process attempts to receive a message from a crashed endpoint, then the process detects the crash and follows its crash handling branch  $P'$ . We now show an example of rule [R- $\zeta\oplus$ ]; more examples can be found in § C.

► **Example 3.** Processes  $P = s[\mathbf{p}][\mathbf{q}]\oplus\mathbf{m}'\langle s[\mathbf{r}] \rangle.s[\mathbf{p}][\mathbf{r}]\&\mathbf{m}(x)$  and  $Q = s[\mathbf{q}][\mathbf{p}]\&\mathbf{m}'(x).x[\mathbf{p}]\oplus\mathbf{m}(42)$  communicate on a session  $s$ ;  $P$  uses  $s[\mathbf{p}]$  to send  $s[\mathbf{r}]$  to role  $\mathbf{q}$ ;  $Q$  uses  $s[\mathbf{q}]$  to receive it, then sends a message to role  $\mathbf{p}$  via  $s[\mathbf{r}]$ . Suppose that  $P$  crashes before sending: this gives rise to

the reduction (by rule  $[\mathbb{R}\text{-}\cancel{\oplus}]$ )  $(\nu s)(P \mid Q) \rightarrow (\nu s)(s[\mathbb{p}] \cancel{\downarrow} \mid s[\mathbb{r}] \cancel{\downarrow} \mid Q)$ . Observe that  $s[\mathbb{p}]$  and  $s[\mathbb{r}]$ , which were held by  $P$ , are now crashed.

### 3 Multiparty Session Types with Crashes

In this section, we present a generalised type system for our multiparty session  $\pi$ -calculus (introduced in Def. 1). As in standard MPST, we assign session types to channel endpoints; we show the syntax of our types in §3.1, where our key additions are *crash handling branches*, and a new type **stop** for crashed endpoints. In §3.2, we give a labelled transition system (LTS) semantics to typing contexts, to represent the behaviour of a collection of types.

Unlike classic MPST, our type system is generalised in the style of [29], hence it has *no* global types; rather, it uses a *safety property* formalising the *minimum* requirement for a typing context to ensure *subject reduction* (and thus, type safety). In this paper, such a safety property is defined in §3.3: unlike previous work, the property accounts for potential crashes, and supports explicit (and optional) reliability assumptions. We show typing rules in §3.4, and the main properties of the typing system: subject reduction (Thm. 11) and session fidelity (Thm. 15) in §3.5. Finally, we demonstrate how we can infer runtime process properties from typing contexts in §3.6.

#### 3.1 Types

A *session type* describes how a process is expected to use a communication channel to send/receive messages to/from other roles involved in a multiparty session. We formalise the syntax of session types in Def. 4, where we add the **stop** type to their standard syntax [29].

► **Definition 4** (Types). *Our types include both basic types and session types:*

$$\begin{array}{ll}
 B ::= \text{int} \mid \text{bool} \mid \text{real} \mid \text{unit} \mid \dots & \text{(basic types)} \\
 S ::= B \mid T & \text{(basic type or session type)} \\
 T ::= \mathbb{p}\&\{\mathbb{m}_i(S_i).T_i\}_{i \in I} \mid \mathbb{p}\oplus\{\mathbb{m}_i(S_i).T_i\}_{i \in I} & \text{(external or internal choice, with } I \neq \emptyset\text{)} \\
 \quad \mid \mu\mathbb{t}.T \mid \mathbb{t} \mid \text{end} & \text{(recursion, type variable, or termination)} \\
 U ::= T \mid \text{stop} & \text{(session type or crash type)}
 \end{array}$$

In internal and external choices, the index set  $I$  must be non-empty, and labels  $\mathbb{m}_i$  must be pairwise distinct. Types are always closed (i.e. each recursion variable  $\mathbb{t}$  is bound under a  $\mu\mathbb{t} \dots$ ) and recursion variables are guarded, i.e. they can only appear under an internal/external choice (e.g.  $\mu\mathbb{t}.\mu\mathbb{t}'.\mathbb{t}$  is not a valid type). For brevity, we may omit the payload type **unit** and the trailing **end**: e.g.  $\mathbb{p}\oplus\mathbb{m}_1.\mathbb{r}\&\mathbb{m}_2$  is shorthand for  $\mathbb{p}\oplus\mathbb{m}_1(\text{unit}).\mathbb{r}\&\mathbb{m}_2(\text{unit}).\text{end}$ .

The internal choice (selection) type  $\mathbb{p}\oplus\{\mathbb{m}_i(S_i).T_i\}_{i \in I}$  denotes *sending* a message  $\mathbb{m}_i$  (by picking some  $i \in I$ ) with a payload of type  $S_i$  to role  $\mathbb{p}$ , and then continue the protocol as  $T_i$ . Dually, the external choice (branching) type  $\mathbb{p}\&\{\mathbb{m}_i(S_i).T_i\}_{i \in I}$  denotes *receiving* a message  $\mathbb{m}_i$  (for any  $i \in I$ ) with a payload of type  $S_i$  from role  $\mathbb{p}$ , and then continue as  $T_i$ . The type **end** indicates that a session endpoint should not be used for further communications.

**Crashes and Crash Detection.** The key novelty of Def. 4 is the new type **stop** describing a crashed session endpoint. Similarly to Def. 1, we also introduce a distinguished message label **crash** for crash handling in external choices. For example, recall the types in §1:

- the type  $\mathbb{q}\&\{\text{res}.T, \text{crash}.T'\}$  means that we expect a **res**ponse message from role  $\mathbb{q}$ , but if we detect that  $\mathbb{q}$  has crashed, then the protocol continues along the handling branch  $T'$ ;
- the type  $\mathbb{q}\&\text{crash}.T$  denotes a “pure” crash recovery behaviour: we are not communicating with  $\mathbb{q}$ , but the recovery protocol  $T$  is activated whenever we detect that  $\mathbb{q}$  has crashed.



$$\begin{array}{c}
\frac{k \in I}{s[p]:q \oplus \{m_i(S_i).T_i\}_{i \in I} \xrightarrow{s[p]:q \oplus m_k(S_k)} s[p]:T_k} [\Gamma-\oplus] \quad \frac{k \in I}{s[p]:q \& \{m_i(S_i).T_i\}_{i \in I} \xrightarrow{s[p]:q \& m_k(S_k)} s[p]:T_k} [\Gamma-\&] \\
\frac{\Gamma_1 \xrightarrow{s[p]:q \oplus m(S)} \Gamma'_1 \quad \Gamma_2 \xrightarrow{s[q]:p \& m(S')} \Gamma'_2 \quad S \leq S'}{\Gamma_1, \Gamma_2 \xrightarrow{s[p]:q \oplus m} \Gamma'_1, \Gamma'_2} [\Gamma-\oplus \&] \quad \frac{s[p]:T\{\mu t.T/t\} \xrightarrow{\alpha} \Gamma' \quad \Gamma \xrightarrow{\alpha} \Gamma'}{s[p]:\mu t.T \xrightarrow{\alpha} \Gamma'} [\Gamma-\mu] \quad \frac{\Gamma \xrightarrow{\alpha} \Gamma' \quad \Gamma, c:U \xrightarrow{\alpha} \Gamma', c:U}{\Gamma, c:U \xrightarrow{\alpha} \Gamma', c:U} [\Gamma-,] \\
\frac{T \not\leq \text{end}}{s[p]:T \xrightarrow{s[p]\not} s[p]:\text{stop}} [\Gamma-\not] \quad \frac{}{s[p]:\text{stop} \xrightarrow{s[p]\text{stop}} s[p]:\text{stop}} [\Gamma-\text{stop}] \quad \frac{\Gamma \xrightarrow{\alpha} \Gamma'}{\Gamma, x:B \xrightarrow{\alpha} \Gamma', x:B} [\Gamma-,B] \\
\frac{\Gamma_1 \xrightarrow{s[q]:p \& \text{crash}} \Gamma'_1 \quad \Gamma_2 \xrightarrow{s[p]\text{stop}} \Gamma'_2}{\Gamma_1, \Gamma_2 \xrightarrow{s[q] \odot p} \Gamma'_1, \Gamma'_2} [\Gamma-\odot] \quad \frac{\Gamma_1 \xrightarrow{s[p]:q \oplus m(S)} \Gamma'_1 \quad \Gamma_2 \xrightarrow{s[q]\text{stop}} \Gamma'_2}{\Gamma_1, \Gamma_2 \xrightarrow{s[p]:q \oplus m} \Gamma'_1, \Gamma'_2} [\Gamma-\not m]
\end{array}$$

■ **Figure 3** Typing context semantics.

Since crash messages *cannot* be crafted by any role in a session (see Def. 1), we postulate that the `crash` message label cannot appear in internal choice types.

**Session Subtyping.** We use a subtyping relation  $\leq$  that is mostly standard: a subtype can have wider internal choices and narrower external choices w.r.t. a supertype. To correctly support crash handling, we apply two changes: (1) we add the relation `stop`  $\leq$  `stop`, and (2) we treat external choices with a singleton `crash` branch in a special way: they represent a “pure” crash recovery protocol (as outlined above), hence we do not allow the supertype to have more input branches. This way, a “pure” crash recovery type can only be implemented by a “pure” crash recovery process (with a singleton crash detection branch); such processes are treated specially by the properties in § 3.6. For the complete definition of  $\leq$ , see § B.

## 3.2 Typing Contexts and their Semantics

Before introducing the typing rules for our calculus (in § 3.4), we first formalise typing contexts (Def. 5) and their semantics (Def. 7).

► **Definition 5** (Typing Contexts).  $\Theta$  denotes a partial mapping from process variables to  $n$ -tuples of types, and  $\Gamma$  denotes a partial mapping from channels to types. Their syntax is:

$$\Theta ::= \emptyset \mid \Theta, X:S_1, \dots, S_n \quad \Gamma ::= \emptyset \mid \Gamma, x:S \mid \Gamma, s[p]:U$$

The context composition  $\Gamma_1, \Gamma_2$  is defined iff  $\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset$ . We write  $s \notin \Gamma$  iff  $\forall p : s[p] \notin \text{dom}(\Gamma)$  (i.e. session  $s$  does not occur in  $\Gamma$ ). We write  $\Gamma \leq \Gamma'$  iff  $\text{dom}(\Gamma) = \text{dom}(\Gamma')$  and  $\forall c \in \text{dom}(\Gamma) : \Gamma(c) \leq \Gamma'(c)$ .

Unlike typical session typing systems, our Def. 5 allows a session endpoint  $s[p]$  to have either a session type  $T$ , or the crash type `stop`. We equip our typing contexts with a labelled transition system (LTS) semantics (in Def. 7) using the labels in Def. 6.

► **Definition 6** (Transition Labels). Let  $\alpha$  denote a transition label having the form:

$$\begin{array}{l}
\alpha ::= s[p]:q \& m(S) \quad (\text{in session } s, \text{ p receives message } m(S) \text{ from } q; \text{ we omit } S \text{ if } S = \text{unit}) \\
\quad \mid s[p]:q \oplus m(S) \quad (\text{in session } s, \text{ p sends message } m(S) \text{ to } q; \text{ we omit } S \text{ if } S = \text{unit}) \\
\quad \mid s[p][q]m \quad (\text{in session } s, \text{ message } m \text{ is transmitted from } p \text{ to } q) \\
\quad \mid s[p]\not \quad (\text{in session } s, \text{ p crashes}) \\
\quad \mid s[p] \odot q \quad (\text{in session } s, \text{ p has detected that } q \text{ has crashed}) \\
\quad \mid s[p]\text{stop} \quad (\text{in session } s, \text{ p has stopped due to a crash})
\end{array}$$

► **Definition 7** (Typing Context Semantics). *The typing context transition  $\xrightarrow{\alpha}$  is defined in Fig. 3. We write  $\Gamma \xrightarrow{\alpha}$  iff  $\Gamma \xrightarrow{\alpha} \Gamma'$  for some  $\Gamma'$ . We define the two reductions  $\rightarrow$  and  $\rightarrow_{\downarrow} \setminus s; \mathcal{R}$  (where  $s$  is a session, and  $\mathcal{R}$  is a set of roles) as follows:*

- $\Gamma \rightarrow \Gamma'$  holds iff  $\Gamma \xrightarrow{s[p][q]m} \Gamma'$  or  $\Gamma \xrightarrow{s[q] \odot p} \Gamma'$  (for some  $s, p, q, m$ ). This means that  $\Gamma$  can advance via message transmission or crash detection, but it cannot advance by crashing one of its entries. We write  $\Gamma \rightarrow$  iff  $\Gamma \rightarrow \Gamma'$  for some  $\Gamma'$ , and  $\Gamma \not\rightarrow$  for its negation (i.e. there is no  $\Gamma'$  such that  $\Gamma \rightarrow \Gamma'$ ), and  $\rightarrow^*$  for the reflexive and transitive closure of  $\rightarrow$ ;
- $\Gamma \rightarrow_{\downarrow} \setminus s; \mathcal{R} \Gamma'$  holds iff  $\Gamma \xrightarrow{\alpha} \Gamma'$  with  $\alpha \in \{s[q][r]m, s[q] \odot r, s[p] \downarrow \mid p, q, r \in \mathfrak{R}, p \notin \mathcal{R}\}$ . This means that  $\Gamma$  can advance via message transmission or crash detection on session  $s$ , involving any roles  $q$  and  $r$ . (Recall that  $\mathfrak{R}$  is the set of all roles.) Moreover,  $\Gamma$  can advance by crashing one of its entries  $s[p]$  – unless  $p \in \mathcal{R}$ , which means that  $p$  is assumed to be reliable. We write  $\Gamma \rightarrow_{\downarrow}^* \setminus s; \mathcal{R} \Gamma'$  iff  $\Gamma \rightarrow_{\downarrow} \setminus s; \mathcal{R} \Gamma'$  for some  $\Gamma'$ , and  $\Gamma \not\rightarrow_{\downarrow} \setminus s; \mathcal{R}$  for its negation, and  $\rightarrow_{\downarrow}^* \setminus s; \mathcal{R}$  as the reflexive and transitive closure of  $\rightarrow_{\downarrow} \setminus s; \mathcal{R}$ . We write  $\Gamma \rightarrow_{\downarrow} \setminus s; \emptyset \Gamma'$  iff  $\Gamma \rightarrow_{\downarrow} \setminus s; \emptyset \Gamma'$  for some  $s$  (i.e.  $\Gamma$  may advance by crashing any role on any session).

Def. 7 subsumes the standard typing context reductions [29, Def. 2.8]. Rule  $[\Gamma-\oplus]$  (resp.  $[\Gamma-\&]$ ) says that an entry can perform an output (resp. input) transition. Rule  $[\Gamma-\oplus\&]$  synchronises matching input/output transitions, provided that the payloads are compatible by subtyping; as a result, the context advances via a message transmission label  $s[p][q]m$ . Other standard rules are  $[\Gamma-\mu]$  for recursion, and  $[\Gamma-]$  and  $[\Gamma, B]$  for reductions in a larger context.

The key innovations are the (highlighted) rules modelling crashes and crash detection. By rule  $[\Gamma-\downarrow]$ , an entry can crash and become **stop** at any time (unless it is already **ended** or **stopped**); then, by rule  $[\Gamma-\text{stop}]$ , it keeps signalling that it is crashed, with label  $s[p]\text{stop}$ .

Rule  $[\Gamma-\odot]$  models crash detection and handling: if  $s[p]$  signals that it has crashed and stopped, another entry  $s[q]$  can then take its **crash** handling branch (part of an external choice from  $p$ ). This corresponds to the process reduction rule  $[\mathbb{R}-\odot]$  for crash detection.

Finally, rule  $[\Gamma-\downarrow m]$  models the case where the entry  $s[p]$  is sending a message  $m(S)$  to a crashed  $s[q]$ : this yields a transmission label  $s[p][q]m$ , and  $p$  continues – although the sent message is not actually received by crashed  $q$ . This corresponds to the process reduction rule  $[\mathbb{R}-\downarrow m]$  where a process sends a message to a crashed endpoint, and cannot detect its crash.

### 3.3 Typing Context Safety

To ensure type safety (Cor. 12), i.e. well-typed processes do not result in **errors**, we define a safety property  $\varphi(\cdot)$  (Def. 8) as a predicate on typing contexts  $\Gamma$ . The safety property  $\varphi$  is the key feature of generalised MPST systems [29, Def. 4.1]; in this work, we extend its definition in two crucial ways: (1) we support crashes and crash detection, and (2) we make the property parametric upon a (possibly empty) set of *reliable* roles  $\mathcal{R}$ , thus introducing *optional reliability assumptions* about roles in a session that never fail.

► **Definition 8** (Typing Context Safety). *Given a set of reliable roles  $\mathcal{R}$  and a session  $s$ , we say that  $\varphi$  is an  $(s; \mathcal{R})$ -safety property of typing contexts iff, whenever  $\varphi(\Gamma)$ , we have:*

$$\begin{aligned} [\mathbb{S}-\oplus\&] \quad & \Gamma \xrightarrow{s[p]:q\oplus m(S)} \text{ and } \Gamma \xrightarrow{s[q]:p\& m'(S')} \text{ implies } \Gamma \xrightarrow{s[p][q]m}; \\ [\mathbb{S}-\downarrow\&] \quad & \Gamma \xrightarrow{s[p]\text{stop}} \text{ and } \Gamma \xrightarrow{s[q]:p\& m(S)} \text{ implies } \Gamma \xrightarrow{s[q] \odot p}; \\ [\mathbb{S}-\rightarrow_{\downarrow}] \quad & \Gamma \rightarrow_{\downarrow} \setminus s; \mathcal{R} \Gamma' \text{ implies } \varphi(\Gamma'). \end{aligned}$$

We say  $\Gamma$  is  $(s; \mathcal{R})$ -safe, written  $\text{safe}(s; \mathcal{R}, \Gamma)$ , if  $\varphi(\Gamma)$  holds for some  $(s; \mathcal{R})$ -safety property  $\varphi$ . We say  $\Gamma$  is safe, written  $\text{safe}(\Gamma)$ , if  $\varphi(\Gamma)$  holds for some property  $\varphi$  which is an  $(s; \emptyset)$ -safety property for all sessions  $s$  occurring in  $\text{dom}(\Gamma)$ .



$$\begin{array}{c}
\frac{\Theta(X) = S_1, \dots, S_n}{\Theta \vdash X:S_1, \dots, S_n} \text{ [T-X]} \quad \frac{v \in B}{\emptyset \vdash v:B} \text{ [T-B]} \quad \frac{\forall i \in 1..n \ S_i \text{ is basic or } c_i:S_i \vdash c_i:\text{end}}{\text{end}(c_1:S_1, \dots, c_n:S_n)} \text{ [T-end]} \\
\frac{\Theta \vdash X:S_1, \dots, S_n \quad \text{end}(\Gamma_0) \quad \forall i \in 1..n \ \Gamma_i \vdash d_i:S_i \quad S_i \not\leq \text{end}}{\Theta \cdot \Gamma_0, \Gamma_1, \dots, \Gamma_n \vdash X\langle d_1, \dots, d_n \rangle} \text{ [T-CALL]} \\
\frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}} \text{ [T-0]} \quad \frac{\Theta, X:S_1, \dots, S_n \cdot x_1:S_1, \dots, x_n:S_n \vdash P \quad \Theta, X:S_1, \dots, S_n \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \text{def } X(x_1:S_1, \dots, x_n:S_n) = P \text{ in } Q} \text{ [T-def]} \\
\frac{\Gamma_1 \vdash c:\mathbf{q}\&\{m_i(S_i).T_i\}_{i \in I} \quad \forall i \in I \ \Theta \cdot \Gamma, y_i:S_i, c:T_i \vdash P_i}{\Theta \cdot \Gamma, \Gamma_1 \vdash c[\mathbf{q}]\&\{m_i(y_i).P_i\}_{i \in I}} \text{ [T-\&]} \quad \frac{\Theta \cdot \Gamma_1 \vdash P_1 \quad \Theta \cdot \Gamma_2 \vdash P_2}{\Theta \cdot \Gamma_1, \Gamma_2 \vdash P_1 \mid P_2} \text{ [T-|]} \\
\frac{\Gamma_1 \vdash c:\mathbf{q}\oplus\{m(S).T\} \quad \Gamma_2 \vdash d:S \quad S \not\leq \text{end} \quad \Theta \cdot \Gamma, c:T \vdash P}{\Theta \cdot \Gamma, \Gamma_1, \Gamma_2 \vdash c[\mathbf{q}]\oplus m\langle d \rangle.P} \text{ [T-\oplus]} \quad \frac{S \leq S'}{c:S \vdash c:S'} \text{ [T-SUB]} \\
\frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma, s[\mathbf{p}]:\text{stop} \vdash s[\mathbf{p}]\not\downarrow} \text{ [T-\not\downarrow]} \quad \frac{\Gamma' = \{s[\mathbf{p}]:T_p\}_{p \in I} \quad \varphi(\Gamma') \quad s \notin \Gamma \quad \Theta \cdot \Gamma, \Gamma' \vdash P}{\Theta \cdot \Gamma \vdash (\nu s:\Gamma') P} \text{ [T-\nu]}
\end{array}$$

■ **Figure 4** Typing rules for processes;  $\varphi$  in [T- $\nu$ ] is an  $(s; \mathcal{R})$ -safety property, for some  $\mathcal{R}$ .

By Def. 8, safety is a *coinductive* property [28]: fix  $s$  and  $\mathcal{R}$ ,  $(s; \mathcal{R})$ -safe is the largest  $(s; \mathcal{R})$ -safety property, i.e. the union of all  $(s; \mathcal{R})$ -safety properties; to prove that some  $\Gamma$  is  $(s; \mathcal{R})$ -safe, we must find a property  $\varphi$  such that  $\Gamma \in \varphi$ , and prove that  $\varphi$  is an  $(s; \mathcal{R})$ -safety property. Intuitively, we can construct such  $\varphi$  (if it exists) as the set containing  $\Gamma$  and all its reductums (via transition  $\rightarrow_{\not\downarrow}^* \lambda_{s; \mathcal{R}}$ ), and checking whether all elements of  $\varphi$  satisfy all clauses of Def. 8. By clause [S- $\oplus\&$ ], whenever two roles  $\mathbf{p}$  and  $\mathbf{q}$  attempt to communicate, the communication must be possible, i.e. the receiver  $\mathbf{q}$  must support all output messages of sender  $\mathbf{p}$ , with compatible payload types (by rule [T- $\oplus\&$ ] in Fig. 3). For “pure” crash recovery types (with a singleton **crash** handling branch) there would not be corresponding sender, so this clause holds trivially. Clause [S- $\not\downarrow\&$ ] states that if a role  $\mathbf{q}$  receives from a crashed role  $\mathbf{p}$ , then  $\mathbf{q}$  must have a **crash** handling branch. Clause [S- $\rightarrow_{\not\downarrow}$ ] states that any typing context  $\Gamma'$  that  $\Gamma$  transitions to (on session  $s$ ) must also be in  $\varphi$  (hence,  $\Gamma'$  must also be  $(s; \mathcal{R})$ -safe); notice that, by using transition  $\rightarrow_{\not\downarrow} \lambda_{s; \mathcal{R}}$ , we ignore crashes  $s[\mathbf{p}]\not\downarrow$  of any reliable role  $\mathbf{p} \in \mathcal{R}$ .

► **Example 9.** Consider the simple DNS scenario from § 1, its types  $T'_p$ ,  $T'_q$  and  $T'_r$ , and the typing context  $\Gamma = s[\mathbf{p}]:T'_p, s[\mathbf{q}]:T'_q, s[\mathbf{r}]:T'_r$ . We know, and can verify, that  $\Gamma$  is  $(s; \{\mathbf{p}, \mathbf{r}\})$ -safe by checking its reductions. For example, for the case where  $\mathbf{q}$  crashes immediately, we have:  $\Gamma \rightarrow_{\not\downarrow} \lambda_{s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:T'_p, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:T'_r \rightarrow_{\not\downarrow}^* \lambda_{s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\text{end}, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:\text{end}$  and each reductum satisfies all clauses of Def. 8. Full reductions are available in § C, Ex. 23.

### 3.4 Typing Rules

Our type system uses two kinds of typing contexts (introduced in Def. 5):  $\Theta$  to assign an  $n$ -tuple of types to each process variable  $X$  (one type per argument), and  $\Gamma$  to map variables to payload types (basic types or session types), and channels with roles to session types or the **stop** type. Together, they are used in judgements of the form:

$$\Theta \cdot \Gamma \vdash P \quad (\text{with } \Theta \text{ omitted when empty})$$

which reads, “given the process types in  $\Theta$ ,  $P$  uses its variables and channels *linearly* according to  $\Gamma$ .” This *typing judgement* is defined by the rules in Fig. 4, where, for convenience, we type-annotate channels bound by process definitions and restrictions.

The main innovations in Fig. 4 are rules [T- $\nu$ ] and [T- $\not\downarrow$ ] (**highlighted**). Rule [T- $\nu$ ] utilises a safety property  $\varphi$  (Def. 8) to validate session restrictions, taking into account crashes and

crash handling, and any reliable role assumption in the (possibly empty) set  $\mathcal{R}$ . The rule can be instantiated by choosing a set  $\mathcal{R}$  and safety property  $\varphi$  (e.g. among the stronger properties presented in Def. 18 later on). Rule  $[\text{T-}\zeta]$  types crashed session endpoints as **stop**.

The rest of the rules in Fig. 4 are mostly standard.  $[\text{T-X}]$  looks up process variables.  $[\text{T-B}]$  types a value  $v$  if it belongs to a basic type  $B$ .  $[\text{T-SUB}]$  holds for a singleton typing context  $c:S$ , and applies subtyping when assigning a type  $S'$  to a variable or channel  $c$ .  $[\text{T-end}]$  defines a predicate  $\text{end}(\cdot)$  on typing contexts, indicating all endpoints are terminated – it is used in  $[\text{T-0}]$  for typing an inactive process  $\mathbf{0}$ , and in  $[\text{T-}\zeta]$  for crashed endpoints.  $[\text{T-}\oplus]$  and  $[\text{T-}\&]$  assign selection and branching types to channels used by selection and branching processes. Minor changes w.r.t. standard session types are the clauses “ $S \not\leq \text{end}$ ” in rules  $[\text{T-}\oplus]$  and  $[\text{T-CALL}]$ : they forbid sending or passing **end**-typed channels, while allowing sending/passing channels and data of any other type.<sup>1</sup> Rules  $[\text{T-def}]$  and  $[\text{T-CALL}]$  handle recursive processes declarations and calls.  $[\text{T-}]$  *linearly* splits the typing context into two, one for typing each sub-process.

### 3.5 Subject Reduction and Session Fidelity

We present our key results on typed processes: *subject reduction* and *session fidelity* (Thms. 11 and 15). A main feature of our theory is that our results explicitly account for the *spectrum* of optional reliability assumptions used during typing.

- On one end of the spectrum, our results hold without any reliability assumption: any process and session endpoint may crash at any time. This is obtained if, for each  $\Gamma$  used during typing, we assume  $\text{safe}(\Gamma)$  (Def. 8), with no reliable roles.
- At the other end of the spectrum, we recover the classic MPST results by assuming that all roles in all sessions are reliable – i.e. if for each  $\Gamma$  used during typing, and for all  $s \in \Gamma$ , we assume  $\text{safe}(s; \mathcal{R}_s, \Gamma)$  with  $\mathcal{R}_s = \{\mathbf{p} \mid s[\mathbf{p}] \in \text{dom}(\Gamma)\}$ .

*Subject reduction* (Thm. 11 below) states that if a well-typed process  $P$  reduces to  $P'$ , then the reduction is simulated by its typing context  $\Gamma$ , provided that the reliability assumptions embedded in  $\Gamma$  hold when  $P$  reduces. In other words, if a channel endpoint  $s[\mathbf{p}]$  occurring in  $P$  is assumed reliable in  $\Gamma$ , then  $P$  should *not* crash  $s[\mathbf{p}]$  while reducing; any other reduction of  $P$  (including those that crash other session endpoints) are type-safe. To formalise this idea, we define *reliable process reduction*  $\rightarrow_{\zeta \setminus s; \mathcal{R}}$  as a subset of  $P$ 's reductions. We also define *assumption-abiding reduction*  $\checkmark$  to enforce reliable process reductions across nested sessions.

► **Definition 10** (Reliable Process Reductions and Assumption-Abiding Reductions). *The reliable process reduction  $\rightarrow_{\zeta \setminus s; \mathcal{R}}$  is defined as follows:*

$$\frac{P \rightarrow P' \quad \forall \mathbf{p} \in \mathcal{R} : \nexists R : P' \equiv R \mid s[\mathbf{p}]\zeta}{P \rightarrow_{\zeta \setminus s; \mathcal{R}} P'}$$

Assume  $\Theta \cdot \Gamma \vdash P$  where, for each  $s \in \Gamma$ , there is a set of reliable roles  $\mathcal{R}_s$  such that  $\text{safe}(s; \mathcal{R}_s, \Gamma)$ . We define the *assumption-abiding reduction*  $\checkmark$  such that  $P \checkmark P'$  holds when: (1)  $P \rightarrow_{\zeta \setminus s; \mathcal{R}_s} P'$  for all  $s \in \Gamma$ ; and (2) if  $P \equiv (\nu s' : \Gamma_{s'}) Q$  (for some  $s', \Gamma_{s'}, Q$ ) and  $P' \equiv (\nu s') Q'$  and  $Q \rightarrow Q'$ , then  $\exists \mathcal{R}'$  such that  $\text{safe}(s'; \mathcal{R}', \Gamma_{s'})$  and  $Q \rightarrow_{\zeta \setminus s'; \mathcal{R}'} Q'$ . We write  $\checkmark^+ / \checkmark^*$  for the transitive / reflexive-transitive closure of  $\checkmark$ .

Hence, when  $P \rightarrow_{\zeta \setminus s; \mathcal{R}} P'$  holds, none of the session endpoints  $s[\mathbf{p}]$  (where  $\mathbf{p}$  is a reliable role in set  $\mathcal{R}$ ) are crashed in  $P'$ . When  $P$  is well-typed, the reduction  $P \checkmark P'$  covers all (and

<sup>1</sup> This restriction is needed for Thms. 11 and 20. It does not limit the expressiveness of our typed calculus, since sending an **end**-typed channel (not usable for communication) amounts to sending a basic value.

only) the reductions of  $P$  that do not violate any reliability assumption used for deriving  $\Theta \cdot \Gamma \vdash P$ ; notice that we use congruence  $\equiv$  to quantify over all restricted sessions in  $P$  and ensure their reductions respect all reliability assumptions in their typing, by  $[\text{T-}\nu]$  in Fig. 4.

We can now use  $\overset{\checkmark}{\rightarrow}$  to state our subject reduction result. Its proof is available in [3].

► **Theorem 11** (Subject Reduction). *Assume  $\Theta \cdot \Gamma \vdash P$  where  $\forall s \in \Gamma : \exists \mathcal{R}_s : \text{safe}(s; \mathcal{R}_s, \Gamma)$ . If  $P \overset{\checkmark}{\rightarrow} P'$ , then  $\exists \Gamma'$  such that  $\Gamma \rightarrow_{\checkmark}^* \Gamma'$ , and  $\forall s \in \Gamma' : \text{safe}(s; \mathcal{R}_s, \Gamma')$ , and  $\Theta \cdot \Gamma' \vdash P'$ .*

► **Corollary 12** (Type Safety). *Assume  $\emptyset \cdot \emptyset \vdash P$ . If  $P \overset{\checkmark}{\rightarrow}^* P'$ , then  $P'$  has no error.*

► **Example 13** (Subject reduction). Take the DNS example (§1) and consider the process acting as the (unreliable) role  $\mathbf{q}$ :  $P_{\mathbf{q}} = s[\mathbf{q}][\mathbf{p}] \& \text{req}.s[\mathbf{q}][\mathbf{p}] \oplus \text{res}.\mathbf{0}$ . Using type  $T'_{\mathbf{q}}$  from the same example, can type  $P_{\mathbf{q}}$  with the typing context  $\Gamma_{\mathbf{q}} = s[\mathbf{q}]:T'_{\mathbf{q}}$ . Following a crash reduction via  $[\text{R-}\&]$ , the process evolves as  $P_{\mathbf{q}} \rightarrow P'_{\mathbf{q}} = s[\mathbf{q}]\checkmark$ . Observe that the typing context  $\Gamma_{\mathbf{q}}$  can reduce to  $\Gamma'_{\mathbf{q}} = s[\mathbf{p}]:\text{stop}$ , via  $[\text{R-}\&]$ ; and by typing rule  $[\text{T-}\&]$ , we can type  $P'_{\mathbf{q}}$  with  $\Gamma'_{\mathbf{q}}$ .

*Session fidelity* states the opposite implication w.r.t. subject reduction: if a process  $P$  is typed by  $\Gamma$ , and  $\Gamma$  can reduce along session  $s$  (possibly by crashing some endpoint of  $s$ ), then  $P$  can reproduce at least one of the reductions of  $\Gamma$  (but maybe not all such reductions, because  $\Gamma$  over-approximates the behaviour of  $P$ ). As a consequence, we can infer  $P$ 's behaviour from  $\Gamma$ 's behaviour, as shown in Thm. 20. This result does *not* hold for all well-typed processes: a well-typed process can loop in a recursion like  $\text{def } X(\dots) = X \text{ in } X$ , or deadlock by suitably interleaving its communications across multiple sessions [13]. Thus, similarly to [29] and most session type works, we prove session fidelity for processes with guarded recursion, and implementing a single multiparty session as a parallel composition of one sub-process per role. Session fidelity is given in Thm. 15 below, by leveraging Def. 14.

► **Definition 14** (from [29]). *Assume  $\emptyset \cdot \Gamma \vdash P$ . We say that  $P$ :*

(1) *has guarded definitions* iff in each process definition in  $P$  of the form  $\text{def } X(x_1:S_1, \dots, x_n:S_n) = Q$  in  $P'$ , for all  $i \in 1..n$ , if  $S_i$  is a session type, then a call  $Y(\dots, x_i, \dots)$  can only occur in  $Q$  as a subterm of  $x_i[\mathbf{q}] \& \{m_j(y_j).P_j\}_{j \in J}$  or  $x_i[\mathbf{q}] \oplus m\langle d \rangle.P''$  (i.e. after using  $x_i$  for input or output);

(2) *only plays role  $\mathbf{p}$  in  $s$ , by  $\Gamma$*  iff: (i)  $P$  has guarded definitions; (ii)  $\text{fv}(P) = \emptyset$ ; (iii)  $\Gamma = \Gamma_0, s[\mathbf{p}]:S$  with  $S \not\leq \text{end}$  and  $\text{end}(\Gamma_0)$ ; (iv) for all subterms  $(\nu s':\Gamma')P'$  in  $P$ ,  $\text{end}(\Gamma')$ .

We say “ $P$  only plays role  $\mathbf{p}$  in  $s$ ” iff  $\exists \Gamma : \emptyset \cdot \Gamma \vdash P$ , and item 2 holds.

Item 1 of Def. 14 formalises guarded recursion for processes. Item 2 identifies a process that plays exactly *one* role on *one* session; clearly, an ensemble of such processes cannot deadlock by waiting for each other on multiple sessions. All our examples satisfy Def. 14(2).

We can now formalise our session fidelity result (Thm. 15). The statement is superficially similar to Thm. 5.4 in [29], but it now includes explicit reliability assumptions for  $\Gamma$ ; it also covers more cases, since our typing contexts and processes can reduce by crashing, handling crashes, or losing messages sent to crashed session endpoints. The proof is available in [3].

► **Theorem 15** (Session Fidelity). *Assume  $\emptyset \cdot \Gamma \vdash P$ , with  $\text{safe}(s; \mathcal{R}, \Gamma)$ ,  $P \equiv \prod_{\mathbf{p} \in I} P_{\mathbf{p}}$ , and  $\Gamma = \bigcup_{\mathbf{p} \in I} \Gamma_{\mathbf{p}}$  such that for each  $P_{\mathbf{p}}$ : (1)  $\emptyset \cdot \Gamma_{\mathbf{p}} \vdash P_{\mathbf{p}}$ , and (2) either  $P_{\mathbf{p}} \equiv \mathbf{0}$ , or  $P_{\mathbf{p}}$  only plays  $\mathbf{p}$  in  $s$ , by  $\Gamma_{\mathbf{p}}$ . Then,  $\Gamma \rightarrow_{\checkmark}^* \Gamma'$  implies  $\exists \Gamma', P'$  such that  $\Gamma \rightarrow_{\checkmark}^* \Gamma'$ ,  $P \overset{\checkmark}{\rightarrow}^+ P'$  and  $\emptyset \cdot \Gamma' \vdash P'$ , with  $\text{safe}(s; \mathcal{R}, \Gamma')$ ,  $P' \equiv \prod_{\mathbf{p} \in I} P'_{\mathbf{p}}$ , and  $\Gamma' = \bigcup_{\mathbf{p} \in I} \Gamma'_{\mathbf{p}}$  such that for each  $P'_{\mathbf{p}}$ : (1)  $\emptyset \cdot \Gamma'_{\mathbf{p}} \vdash P'_{\mathbf{p}}$ , and (2) either  $P'_{\mathbf{p}} \equiv \mathbf{0}$ , or  $P'_{\mathbf{p}}$  only plays  $\mathbf{p}$  in  $s$ , by  $\Gamma'_{\mathbf{p}}$ .*

### 3.6 Statically Verifying Run-Time Properties of Processes with Crashes

We conclude this section by showing how to infer run-time process properties from typing contexts, even in the presence of arbitrary process crashes. The formulations are based on [29, Def. 5.1 & Fig. 5(1)], but (1) we cater for optional assumptions on reliable roles; (2) a successfully-terminated process or typing context may include crashed session endpoints and failover types/processes (like DNS server  $\mathbf{r}$  in §1) that only run after detecting a crash; and (3) non-failover reliable roles terminate by reaching  $\mathbf{0}$  (in processes) or  $\mathbf{end}$  (in types).

Def. 16 formalises several desirable process properties, using the assumption-abiding reduction  $\check{\rightarrow}$  (Def. 10) to embed any assumptions on reliable roles used for typing. The properties are mostly self-explanatory: *deadlock-freedom* means that if a process cannot reduce, then it only contains inactive or crashed sub-processes, or recovery processes attempting to detect others' crashes; *liveness* means that if a process is trying to perform an input or output, then it eventually succeeds (unless it is only attempting to detect others' crashes).

► **Definition 16** (Runtime Process Properties). *Assume  $\emptyset \cdot \Gamma \vdash P$  where,  $\forall s \in \Gamma$ , there is a set of roles  $\mathcal{R}_s$  such that  $\text{safe}(s; \mathcal{R}_s, \Gamma)$ . We say  $P$  is:*

(1) **deadlock-free** iff  $P \check{\rightarrow}^* P' \not\rightarrow$  implies

$$P' \equiv \mathbf{0} \mid \Pi_{i \in I} s_i[\mathbf{p}_i] \not\downarrow \mid \Pi_{j \in J} (\text{def } D_{j,1} \text{ in } \dots \text{ def } D_{j,n_j} \text{ in } s_j[\mathbf{p}_j][\mathbf{q}_j] \& \text{crash}.Q'_j);$$

(2) **terminating** iff it is deadlock-free, and  $\exists j$  finite such that  $\forall n \geq j: P = P_0 \check{\rightarrow} P_1 \check{\rightarrow} \dots \check{\rightarrow} P_n$  implies  $P_n \not\rightarrow$ ;

(3) **never-terminating** iff  $P \check{\rightarrow}^* P'$  implies  $P' \rightarrow$ ;

(4) **live** iff  $P \check{\rightarrow}^* P' \equiv \mathbb{C}[Q]$  implies:

(i) if  $Q = c[\mathbf{q}] \oplus \mathbf{m}(w).Q'$  then  $\exists \mathbb{C}' : P' \rightarrow^* \mathbb{C}'[Q']$ ;

(ii) if  $Q = c[\mathbf{q}] \& \{\mathbf{m}_i(x_i).Q'_i\}_{i \in I}$  where  $\{\mathbf{m}_i \mid i \in I\} \neq \{\text{crash}\}$ , then  $\exists \mathbb{C}', k \in I, w :$   
 $P' \rightarrow^* \mathbb{C}'[Q'_k\{w/x_k\}]$ .

In Def. 18 we formalise the type-level properties corresponding to Def. 16. Type-level liveness means that all pending internal/external choices are eventually fired (via a message transmission or crash detection) – assuming *fairness* (Def. 17, based on *strong fairness of components* [32, Fact 2]) so all enabled message transmissions are eventually performed.

► **Definition 17** (Non-crashing, Fair, Live Paths (adapted from [17, Def. 4.4])). *A **non-crashing path** is a possibly infinite sequence of typing contexts  $(\Gamma_n)_{n \in N}$ , where  $N = \{0, 1, 2, \dots\}$  is a set of consecutive natural numbers, and,  $\forall n \in N, \Gamma_n \rightarrow \Gamma_{n+1}$ .*

*We say that a non-crashing path  $(\Gamma_n)_{n \in N}$  is **fair for session  $s$**  iff,  $\forall n \in N: \Gamma_n \xrightarrow{s[\mathbf{p}][\mathbf{q}]\mathbf{m}} \Gamma_{k+1}$  implies  $\exists k, \mathbf{m}'$  such that  $N \ni k \geq n$ , and  $\Gamma_k \xrightarrow{s[\mathbf{p}][\mathbf{q}]\mathbf{m}'} \Gamma_{k+1}$ .*

*We say that a non-crashing path  $(\Gamma_n)_{n \in N}$  is **live for session  $s$**  iff,  $\forall n \in N:$*

(1)  $\Gamma_n \xrightarrow{s[\mathbf{p}]:\mathbf{q} \oplus \mathbf{m}(S)} \Gamma_{k+1}$  implies  $\exists k, \mathbf{m}'$  such that  $N \ni k \geq n$  and  $\Gamma_k \xrightarrow{s[\mathbf{p}][\mathbf{q}]\mathbf{m}'} \Gamma_{k+1}$ ;

(2)  $\Gamma_n \xrightarrow{s[\mathbf{q}]:\mathbf{p} \& \mathbf{m}(S)} \Gamma_{k+1}$  and  $\mathbf{m} \neq \text{crash}$  implies  $\exists k, \mathbf{m}'$  such that  $N \ni k \geq n$  and  $\Gamma_k \xrightarrow{s[\mathbf{p}][\mathbf{q}]\mathbf{m}'} \Gamma_{k+1}$  or  $\Gamma_k \xrightarrow{s[\mathbf{q}]\odot \mathbf{p}} \Gamma_{k+1}$ .

► **Definition 18** (Typing Context Properties). *Given a session  $s$  and a set of reliable roles  $\mathcal{R}$ , we say  $\Gamma$  is:*

(1)  **$(s; \mathcal{R})$ -deadlock-free** iff  $\Gamma \rightarrow_{\not\downarrow}^* \Gamma' \not\rightarrow$  implies  $\forall s[\mathbf{p}] \in \Gamma : \Gamma(s[\mathbf{p}]) \leq \mathbf{end}$  or  $\Gamma(s[\mathbf{p}]) = \text{stop}$  or  $\exists \mathbf{q} : \Gamma(s[\mathbf{p}]) \leq \mathbf{q} \& \text{crash}.T'$ ;

(2)  **$(s; \mathcal{R})$ -terminating** iff it is deadlock-free, and  $\exists j$  finite such that  $\forall n \geq j: \Gamma = \Gamma_0 \rightarrow_{\not\downarrow} \Gamma_1 \rightarrow_{\not\downarrow} \Gamma_2 \dots \rightarrow_{\not\downarrow} \Gamma_n$  implies  $\Gamma_n \not\rightarrow$ ;

- (3)  $(s; \mathcal{R})$ -never-terminating iff  $\Gamma \rightarrow_{i \setminus s; \mathcal{R}}^* \Gamma'$  implies  $\Gamma' \rightarrow$ ;  
 (4)  $(s; \mathcal{R})$ -live iff  $\Gamma \rightarrow_{i \setminus s; \mathcal{R}}^* \Gamma'$  implies all non-crashing paths starting with  $\Gamma'$  which are fair for session  $s$  are also live for  $s$ .

► **Example 19.** Reliability assumptions  $\mathcal{R}$  can affect typing context properties, e.g. consider:

$$\Gamma = s[\mathbf{p}]:\mu_{\mathbf{p}}.q \oplus \text{ok}.t_{\mathbf{p}}, s[\mathbf{q}]:\mu_{\mathbf{q}}.p \& \{ \text{ok}.t_{\mathbf{q}}, \text{crash}.\mu_{\mathbf{q}}.r \& \{ \text{ok}.t'_{\mathbf{q}}, \text{crash}.\text{end} \} \}, s[\mathbf{r}]:\mu_{\mathbf{r}}.q \oplus \text{ok}.t_{\mathbf{r}}$$

If  $\mathcal{R} = \emptyset$ ,  $\Gamma$  is safe and deadlock-free but *not* live: if  $\mathbf{p}$  does *not* crash,  $\mathbf{r}$ 's *ok* message is never received by  $\mathbf{q}$ . If we have  $\mathcal{R} = \{\mathbf{r}\}$ ,  $\Gamma$  satisfies never-termination. Here, neither liveness nor termination can be satisfied by adding reliability assumptions. More examples in § C, Ex. 24.

We conclude by showing how the type-level properties in Def. 18 allow us to infer the corresponding process properties in Def. 16. The proof is available in [3].

► **Theorem 20** (Verification of Process Properties). *Assume  $\emptyset \cdot \Gamma \vdash P$ , where  $\Gamma$  is  $(s; \mathcal{R})$ -safe,  $P \equiv \Pi_{\mathbf{p} \in I} P_{\mathbf{p}}$ , and  $\Gamma = \bigcup_{\mathbf{p} \in I} \Gamma_{\mathbf{p}}$  such that for each  $P_{\mathbf{p}}$ , we have  $\emptyset \cdot \Gamma_{\mathbf{p}} \vdash P_{\mathbf{p}}$ . Further, assume that each  $P_{\mathbf{p}}$  is either  $\mathbf{0}$  (up to  $\equiv$ ), or only plays  $\mathbf{p}$  in  $s$ , by  $\Gamma_{\mathbf{p}}$ . Then, for all  $\varphi \in \{\text{deadlock-free, terminating, never-terminating, live}\}$ , if  $\Gamma$  is  $(s; \mathcal{R})$ - $\varphi$ , then  $P$  is  $\varphi$ .*

## 4 Verifying Type-Level Properties via Model Checking

In our generalised typing system, we prove subject reduction when a typing context satisfies a safety property (Def. 8); we then give examples of more refined typing context properties (Def. 18) and show how they are inherited by typed processes (Thm. 20). In this section, we highlight a major benefit of our theory: we show how such typing context behavioural properties can be verified using model checkers. We use our typing contexts and their semantics (including crashes and crash handling) as models, and we express our behavioural properties as modal  $\mu$ -calculus formulæ; we then use a model checker (mCRL2 [4]) to verify whether a typing context enjoys a desired property.

**Contexts as Models.** We encode our typing contexts as mCRL2 processes, with LTS semantics that match Def. 7. To embed our optional reliability assumptions, the context encoding reflects the transition relation  $\rightarrow_{i \setminus s; \mathcal{R}}$ , so it never crashes any reliable role in  $\mathcal{R}$ .

**Properties as Formulæ.** A modal  $\mu$ -calculus formula  $\phi$  accepts or rejects a typing context  $\Gamma$  depending on the transition labels  $\Gamma$  can fire while reducing. We write  $\Gamma \models \phi$  when a typing context  $\Gamma$  satisfies  $\phi$ . Actions  $\alpha$  range over transition labels in Def. 6;  $\mathbf{d}$  (for **d**ata) ranges over sessions, roles, message labels, and types. Our formulæ  $\phi$  follow a standard syntax:

$$\phi ::= \top \mid \perp \mid [\alpha]\phi \mid \langle \alpha \rangle \phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \mu \mathbf{Z}.\phi \mid \nu \mathbf{Z}.\phi \mid \mathbf{Z} \mid \forall \mathbf{d}.\phi \mid \exists \mathbf{d}.\phi$$

Truth ( $\top$ ) accepts any  $\Gamma$ ; falsity ( $\perp$ ) accepts no  $\Gamma$ . The box (resp. diamond) modality,  $[\alpha]\phi$  (resp.  $\langle \alpha \rangle \phi$ ), requires that  $\phi$  is satisfied in all cases (resp. some cases) after action  $\alpha$  is fired. The least (resp. greatest) fixed point  $\mu \mathbf{Z}.\phi$  (resp.  $\nu \mathbf{Z}.\phi$ ) allows one to iterate  $\phi$  for a finite (resp. infinite) number of times, where  $\mathbf{Z}$  denotes a variable for iteration. Lastly, the forms  $\phi_1 \Rightarrow \phi_2$ ,  $\forall \mathbf{d}.\phi$ , and  $\exists \mathbf{d}.\phi$  denote implication, and universal and existential quantification.

In Fig. 5 we show the  $\mu$ -calculus formulæ corresponding to our properties in Defs. 8 and 18. Compared to [29], such properties are more complex, since they cater for crashes and crash handling transitions. Recall Def. 8, and take a safety property  $\varphi$ : for  $\varphi(\Gamma)$  to hold, clause  $[\mathbf{s} \rightarrow_i]$  requires that whenever  $\Gamma$  can transition to some  $\Gamma'$  (via  $\rightarrow_{i \setminus s; \mathcal{R}}$ ), then  $\varphi(\Gamma')$  also holds. To represent this clause in modal  $\mu$ -calculus, we use fixed points for possibly infinite paths;

$$\begin{array}{l}
 \text{[}\mu\text{-SAFE] } \Gamma \models \nu Z. \left( \forall s, p, q, m, m', S, S'. \left( \langle s[p]\text{stop} \rangle T \wedge \langle s[q]:p\&m'(S') \rangle T \Rightarrow \langle s[q]\odot p \rangle T \right) \right. \\
 \left. \wedge \left( \langle s[p]:q\oplus m(S) \rangle T \wedge \left( \langle s[q]:p\&m'(S') \rangle T \vee \langle s[q]\text{stop} \rangle T \right) \Rightarrow \langle s[p][q]m \rangle T \right) \wedge \phi_{\rightarrow}(Z) \right) \\
 \text{[}\mu\text{-DF] } \Gamma \models \nu Z. \left( \left( \forall s, p, q, m. [s[p][q]m] \perp \wedge [s[p]\odot q] \perp \Rightarrow \right. \right. \\
 \left. \left. \forall s, p, q, m, S. [s[p]:q\&m(S)] \perp \wedge [s[p]:q\oplus m(S)] \perp \right) \wedge \phi_{\rightarrow}(Z) \right) \\
 \text{[}\mu\text{-TERM] } \Gamma \models \mu Z. \left( \left( \forall s, p, q, m. [s[p][q]m] \perp \wedge [s[p]\odot q] \perp \Rightarrow \right. \right. \\
 \left. \left. \forall s, p, q, m, S. [s[p]:q\&m(S)] \perp \wedge [s[p]:q\oplus m(S)] \perp \right) \wedge \phi_{\rightarrow}(Z) \right) \\
 \text{[}\mu\text{-NTERM] } \Gamma \models \nu Z. \left( \exists s, p, q, m. \langle s[p][q]m \rangle T \vee \langle s[p]\odot q \rangle T \right) \wedge \phi_{\rightarrow}(Z) \\
 \text{[}\mu\text{-LIVE] } \Gamma \models \nu Z. \left( \forall s, p, q. \left( \begin{array}{l} \phi_{in} = \left( \begin{array}{l} \exists m, S. \langle s[q]:p\&m(S) \rangle T \Rightarrow \\ \mu Z'. \left( \begin{array}{l} \exists m. \langle s[p][q]m \rangle T \vee \langle s[q]\odot p \rangle T \\ \vee \exists p', q'. \left( \begin{array}{l} \exists m'. \langle s[p']][q']m' \rangle T \vee \langle s[q']\odot p' \rangle T \\ \wedge \phi'_{\rightarrow}(s, p', q', Z') \end{array} \right) \end{array} \right) \end{array} \right) \right) \\ \wedge \phi_{out} = \forall m. \left( \begin{array}{l} \exists S. \langle s[p]:q\oplus m(S) \rangle T \Rightarrow \\ \mu Z'. \left( \begin{array}{l} \langle s[p][q]m \rangle T \\ \vee \exists p', q'. \left( \begin{array}{l} \exists m'. \langle s[p']][q']m' \rangle T \vee \langle s[q']\odot p' \rangle T \\ \wedge \phi'_{\rightarrow}(s, p', q', Z') \end{array} \right) \end{array} \right) \end{array} \right) \right) \\ \wedge \phi_{\rightarrow}(Z) \end{array} \right) \right)
 \end{array}
 \end{array}$$

■ **Figure 5** Modal  $\mu$ -Calculus Formulæ corresponding to Properties in Defs. 8 and 18, where  $\phi_{\rightarrow}(Z) = \forall s, p, q. \phi'_{\rightarrow}(s, p, q, Z)$ , and  $\phi'_{\rightarrow}(s, p, q, Z) = \forall m. [s[p][q]m]Z \wedge [s[p]\not\downarrow]Z \wedge [s[p]\odot q]Z$ .

in Fig. 5 we write  $\phi_{\rightarrow}(Z)$  for following a fixed point  $Z$  via any transmission, crash,<sup>2</sup> or crash handling actions, and we define it as  $\phi_{\rightarrow}(Z) = \forall s, p, q, m. [s[p][q]m]Z \wedge [s[p]\not\downarrow]Z \wedge [s[p]\odot q]Z$ .

**Safety** ( $[\mu\text{-SAFE}]$ ) requires (in its second implication) that whenever  $\Gamma$  can fire an input action, and either an output or  $s[q]\text{stop}$  action, then  $\Gamma$  can also fire a message transmission,  $s[p][q]m$ . The first implication requires that, if  $\Gamma$  can fire a  $s[p]\text{stop}$  action and an input action  $s[q]:p\&m'(S')$ , then  $\Gamma$  must be capable of firing a crash handling action,  $s[q]\odot p$ .

**Deadlock-Freedom** ( $[\mu\text{-DF}]$ ) requires that, if  $\Gamma$  is unable to reduce further without crashing (via  $\rightarrow$ ), then  $\Gamma$  can hold only **ended** or **stopped** endpoints. The antecedent of  $\Rightarrow$  characterises a context that is unable to reduce (since  $\rightarrow$  only allows for transmissions  $s[p][q]m$  and crash detection  $s[p]\odot q$ ); the consequent forbids the presence of any input  $s[p]:q\&m(S)$  or output  $s[p]:q\oplus m(S)$  transitions. By Def. 7, this means all session endpoints in  $\Gamma$  are **ended** or **stopped**.

**Terminating** ( $[\mu\text{-TERM}]$ ) holds when  $\Gamma$  can reach a terminal configuration (i.e. cannot further reduce via  $\rightarrow$ ) within a *finite* number of steps. Hence, the formula is similar to deadlock-freedom, except that it uses the *least* fixed point ( $\mu Z \dots$ ) to ensure finiteness.

**Never-Terminating** ( $[\mu\text{-NTERM}]$ ) requires that  $\Gamma$  can always keep reducing via  $\rightarrow$  transitions. Therefore, we require some transmission  $s[p][q]m$  or crash detection action  $s[p]\odot q$  to be always fireable, even after some of the non-reliable roles crash.

**Liveness** ( $[\mu\text{-LIVE}]$ ) requires that any enabled input/output action is triggered by a corresponding message transmission or crash detection, within a finite number of steps. For input actions (sub-formula  $\phi_{in}$ ): if an input  $s[q]:p\&m(S)$  is enabled (left of  $\Rightarrow$ ), then, in a finite number of steps ( $\mu Z' \dots$ ) involving *other* roles  $p', q'$ , a transmission  $s[p][q]m'$  or a crash detection  $s[q]\odot p$  can be fired. For output actions, the sub-formula  $\phi_{out}$  is similar. The  $\mu$ -calculus formula embeds fairness (Def. 17) by finding *some* roles  $p', q'$  that, no matter how they interact (sub-formula  $\phi'_{\rightarrow}$ ), lead to the desired transmission or crash detection.

**Tool Implementation and Example.** To verify the properties in Fig. 5, we implement a prototype tool that extends `mpstk` [30] (based on the mCRL2 model checker [4]) with support

<sup>2</sup> Since our typing contexts encoded in mCRL2 produce  $\rightarrow_{\not\downarrow} \setminus s; \mathcal{R}$ -transitions that never crash reliable roles in  $\mathcal{R}$ , our  $\mu$ -calculus formulæ can follow *all* crash transitions; hence, the formulæ do not depend on  $\mathcal{R}$ .



```

s[b1]:s⊕req(Str).s& {quote(Int).b2⊕split(Int).b2& {crash.T1}, crash.b2⊕Tko}
    T1 = s& {rp1.s⊕{ok.T2, Tko}, rp2.T2, rp3.s& {date(Str).end, Tℓ}, Tℓ}
    T2 = s⊕addr(Str).s& {date(Str).end, Tℓ}
s[b2]:s& {quote(Int).T1, Tko, crash.T1}
    T1 = b1& {split(Int).s⊕{ok.s⊕addr(Str).s& {date(Str).end, Tℓ}, Tko}, Tko, crash.s⊕Tko}
s[s]:b1& {req(Str).b1⊕quote(Int).b2⊕quote(Int).b2& {ok.T1, Tko, crash.b1⊕rp1.T2}, crash.b2⊕Tko}
    T1 = b2& {addr(Str).b2⊕date(Str).b2& {crash.b1⊕rp3.T4}, crash.b1⊕rp2.T3}
    T2 = b1& {ok.T3, Tko, Tℓ}    T3 = b1& {addr(Str).T4, Tℓ}    T4 = b1⊕date(Str).end
where: Tℓ = crash.end    Tok = ok.end    Tko = ko.end

```

■ **Figure 6** Two-Buyers protocol extended with crash-handling.

for our crash-stop semantics. The updated tool is available at:

<https://github.com/alcestes/mpstk-crash-stop>

We now illustrate how this new tool helps in writing correct session protocols with crash handling, and briefly discuss its performance.

In the *two-buyers protocol* from MPST literature [19], buyers **b1** and **b2** agree on splitting the cost of buying a book from seller **s**. We tackle this protocol with crashes and *no reliability assumptions*: all roles may crash, and survivors must end the session correctly. The resulting *crash-tolerant two-buyers protocol* (Fig. 6) is much more complex than the one in the literature. In fact, the possibility of crashes introduces a variety of scenarios where different roles may be crashed (or not), hence the protocol needs many `crash` branches. The protocol exhibits two crash-handling patterns: *i*) exiting gracefully, and *ii*) recovery behaviour. The former occurs either when **s** crashes or when **b1** crashes prior to the agreed split. The latter occurs should **b2** crash after the agreed split, whereupon **b2** concludes the transaction if both **b2** and **s** do not crash. This behaviour is activated via a recovery type in  $s[b1]$ , where the labels `rpn` represent the point at which **b2** crashed: `rp1` represents **b2** failing prior to confirmation with **s**; `rp2` corresponds to before the sending of `addr`; and `rp3` prior to receiving the `date`. Overlooking or mishandling some cases is easy; our tool spots such errors, so the protocol can be tweaked until all desired properties hold. We used our tool to verify the protocol: it has 1409 states and 10248 transitions; it is safe, deadlock-free, live, and it is terminating; it is *not* never-terminating. All properties verify within 100ms on a 4.20 GHz Intel Core i7-7700K CPU with 16 GB RAM. More experimental results can be found in §D.

## 5 Related Work, Conclusions, and Future Work

**Previous Work on Failure Handling in Session Types** can be generally classified under two main approaches: *affine* and *coordinator model*. The former adapts session types to allow session endpoints to cease prematurely (e.g. by throwing an exception); the latter assumes reliable process coordination to handle failures.

*Affine failure handling* is first proposed in [24] for a  $\pi$ -calculus with binary sessions (i.e. two roles), and [14] presents a concurrent  $\lambda$ -calculus with binary sessions and exception handling; exceptions are also found in [7,8]. These works model failures at the application level, via throw/catch constructs. Our key innovations are: (1) we model arbitrary failures (e.g. hardware failures); (2) we specify what to do when a failure is detected *at the type level*; (3) we support multiparty sessions; and (4) we seamlessly support handling the crash of a role while handling another role's crash, whereas the *do-catch* constructs cannot be nested.

*Coordinator model approaches* include [1], which extends MPST with *optional blocks* where *default values* are used when communications fail; and [12], which uses synchronisation

points to detect and handle failures. Both need processes to coordinate to handle failures. [33] extends MPST with a *try-handle* construct: a reliable coordinator detects and broadcasts failures, and the remaining processes proceed with failure handling. Unlike these works, we do *not* assume reliable processes, failure broadcasts, or coordination/synchronisation points.

Other papers address failures with different approaches. The recent work [26] annotates global and local types to specify which interactions may fail, and how (process crash, message loss). Their failure model is different from ours; and unlike us, they handle failures by continuing the protocol via *default branches and values*. Instead, our types include **crash** branches defining recovery behaviours that are only executed upon crash detection; further, by nesting such **crash** branches, we can specify different behaviours depending on which roles have crashed. [25] uses an MPST specification to build a dependency graph among running processes, supervise them, and restart them in case of failure. [34] utilises MPST to specify fault-tolerant, event-driven distributed systems, where processes are monitored and restarted if they fail; unlike our work, they require certain reliable roles, but their model tolerates false crash suspicions. More on the theory side, [6] presents a Curry-Howard interpretation of a language with binary session types and internal non-determinism, which is used to model failures (that are propagated to all relevant sessions, similarly to [14, 24]). Process calculi with *localities* have been proposed to model distributed systems with failures [2, 9, 27]; unlike our work, they do not have a typing system to verify failure handling.

**Generalised Multiparty Session Type Systems** (introduced in [29]) depart from “classic” MPST [20] by not requiring top-down syntactic notions of protocol correctness (global types, projection, *etc.*); rather, they check behavioural predicates (safety, liveness, *etc.*) against (local) session types. [18] adopts the approach to model actor systems with explicit connections in their types [21]. By adopting this general framework, we support protocols not representable as global types in classic MPST (e.g. DNS in § 1, two-buyers in § 4, and all examples in § D, excepting **Adder**).

**Model Checking Behavioural Types.** [10] develops a behavioural type system for the  $\pi$ -calculus, and check LTL formulæ against such types. In [22], the type system combines typing and local analyses, with liveness properties verified via model checking. A similar approach is introduced in [29] for MPST. Regarding applications, [15, 23] verify behavioural types extracted from Go source code; and in [31], the **Effpi** Scala library assigns behavioural types to communicating programs. These works use a model checker to validate e.g. liveness through type-level properties, but do not support crashes or crash handling.

**Conclusions and Future Work.** We presented a multiparty session typing system for verifying processes with crash-stop failures. We model crashes and crash handling in a session  $\pi$ -calculus and its typing contexts, and prove type safety, protocol conformance, deadlock freedom and liveness. Our system is generalised in two ways: (1) it supports *optional* reliability assumptions, ranging from fully reliable (as in classic MPST), to fully unreliable (every process may crash); and (2) it is parametric on a behavioural property  $\varphi$  (validated by model checking) which can ensure deadlock-freedom, liveness, *etc. even in presence of crashes*. We also present a prototype implementation of our approach. As future work, we plan to study more crash models (e.g. crash-recover) and types of failure (e.g. link failures). We also plan to study the use of *asynchronous* global types for specifying protocols with failure handling — but unlike [26], we plan to support the type-level specification of dedicated recovery behaviours that are only executed upon crash detection.

## References

- 1 Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2017. doi:10.1007/978-3-319-60225-7\_1.
- 2 Roberto M. Amadio. An asynchronous model of locality, failure and process mobility. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models, Second International Conference, COORDINATION '97, Berlin, Germany, September 1-3, 1997, Proceedings*, volume 1282 of *Lecture Notes in Computer Science*, pages 374–391. Springer, 1997. doi:10.1007/3-540-63383-9\_92.
- 3 Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. Technical report, 2022. doi:https://doi.org/10.48550/arXiv.2207.02015.
- 4 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 Toolset for Analysing Concurrent Systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.
- 5 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. doi:10.1007/978-3-642-15260-3.
- 6 Luís Caires and Jorge A. Pérez. Linearity, Control Effects, and Behavioral Types. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 229–259. Springer, 2017. doi:10.1007/978-3-662-54434-1\_9.
- 7 Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Math. Struct. Comput. Sci.*, 26(2):156–205, 2016. doi:10.1017/S0960129514000164.
- 8 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Interactional Exceptions in Session Types. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 402–417. Springer, 2008. doi:10.1007/978-3-540-85361-9\_32.
- 9 Iaria Castellani. Process algebras with localities. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 945–1045. North-Holland / Elsevier, 2001. doi:10.1016/b978-044482830-9/50033-3.
- 10 Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: Model checking message-passing programs. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 45–57, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/503272.503278.
- 11 Tushar Deepak Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, March 1996. doi:10.1145/226643.226647.
- 12 Tzu-Chun Chen, Malte Viering, Andi Bejleri, Lukasz Ziarek, and Patrick Eugster. A Type Theory for Robust Failure Handling in Distributed Systems. In Elvira Albert and Ivan Lanese, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 2016. doi:10.1007/978-3-319-39570-8\_7.

- 13 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 760, 2015. doi:10.1017/S0960129514000188.
- 14 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.
- 15 Julia Gabet and Nobuko Yoshida. Static Race Detection and Mutex Safety and Liveness for Go Programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:30, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.4.
- 16 Simon Gay and António Ravara. *Behavioural Types: From Theory to Tools*. River Publishers, Series in Automation, Control and Robotics, 2017. doi:10.13052/rp-9788793519817.
- 17 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi:10.1145/3434297.
- 18 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2021.10.
- 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, 2008. Full version in [20]. doi:10.1145/1328438.1328472.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1), 2016. doi:10.1145/2827695.
- 21 Raymond Hu and Nobuko Yoshida. Explicit Connection Actions in Multiparty Session Types. In *FASE*, 2017. doi:10.1007/978-3-662-54494-5\_7.
- 22 Naoki Kobayashi and Davide Sangiorgi. A Hybrid Type System for Lock-Freedom of Mobile Processes. *TOPLAS*, 32(5), 2010. doi:10.1145/1745312.1745313.
- 23 Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A Static Verification Framework for Message Passing in Go Using Behavioural Types. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1137–1148, 2018. doi:10.1145/3180155.3180157.
- 24 Dimitris Mostrous and Vasco T. Vasconcelos. Affine Sessions. *Logical Methods in Computer Science*, Volume 14, Issue 4, November 2018. doi:10.23638/LMCS-14(4:14)2018.
- 25 Rumyana Neykova and Nobuko Yoshida. Let It Recover: Multiparty Protocol-Induced Recovery. In *CC*, 2017. doi:10.1145/3033019.3033031.
- 26 Kirstin Peters, Uwe Nestmann, and Christoph Wagner. Fault-tolerant multiparty session types. In Mohammad Reza Mousavi and Anna Philippou, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 42nd IFIP WG 6.1 International Conference, FORTE 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13273 of *Lecture Notes in Computer Science*, pages 93–113. Springer, 2022. doi:10.1007/978-3-031-08679-3\_7.
- 27 James Riely and Matthew Hennessy. Distributed processes and location failures (extended abstract). In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, volume 1256 of *Lecture Notes in Computer Science*, pages 471–481. Springer, 1997. doi:10.1007/3-540-63165-8\_203.
- 28 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. doi:10.1017/CB09780511777110.

- 29 Alceste Scalas and Nobuko Yoshida. Less is More: Multiparty Session Types Revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. doi:10.1145/3290343.
- 30 Alceste Scalas and Nobuko Yoshida. mpstk: the Multiparty Session Types ToolKit, 2019. Peer-reviewed artifact of [29]. (Latest version available at: <https://alcestes.github.io/mpstk>). doi:10.1145/3291638.
- 31 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying Message-Passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, page 502–516, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3322484.
- 32 Rob van Glabbeek, Peter Höfner, and Ross Horne. Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470531.
- 33 Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu, and Lukasz Ziarek. A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 799–826, Cham, 2018. Springer International Publishing.
- 34 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), Oct 2021. doi:10.1145/3485501.
- 35 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020. doi:10.1145/3428216.

## A Structural Congruence

The structural congruence relation of our MPST  $\pi$ -calculus, mentioned in Fig. 2, is formalised below. These rules are standard, and taken from [29]; the only extension is rule [C-CRASHELIM]. Here,  $\text{fpv}(D)$  is the set of *free process variables* in  $D$ , and  $\text{dpv}(D)$  is the set of *declared process variables* in  $D$ .

$$\begin{aligned}
P \mid Q &\equiv Q \mid P \quad [\text{C-PAR}] & (P \mid Q) \mid R &\equiv P \mid (Q \mid R) \quad [\text{C-ASSOC}] & P \mid \mathbf{0} &\equiv P \quad [\text{C-PARID}] \\
(\nu s) \mathbf{0} &\equiv \mathbf{0} \quad [\text{C-RESELIM}] & (\nu s) (\nu s') P &\equiv (\nu s') (\nu s) P \quad [\text{C-RESVAR}] \\
(\nu s) (P \mid Q) &\equiv P \mid (\nu s) Q \quad \text{if } s \notin \text{fc}(P) \quad [\text{C-RESLIFT}] & (\nu s) (s[\mathbf{p}_0] \zeta \mid \cdots \mid s[\mathbf{p}_n] \zeta) &\equiv \mathbf{0} \quad [\text{C-CRASHELIM}] \\
\text{def } D \text{ in } \mathbf{0} &\equiv \mathbf{0} \quad [\text{C-DEFELIM}] & \text{def } D \text{ in } (\nu s) P &\equiv (\nu s) (\text{def } D \text{ in } P) \quad \text{if } s \notin \text{fc}(D) \quad [\text{C-DEFLIFT}] \\
\text{def } D \text{ in } (P \mid Q) &\equiv (\text{def } D \text{ in } P) \mid Q \quad \text{if } \text{dpv}(D) \cap \text{fpv}(Q) = \emptyset \quad [\text{C-DEFPARLIFT}] \\
\text{def } D \text{ in } (\text{def } D' \text{ in } P) &\equiv \text{def } D' \text{ in } (\text{def } D \text{ in } P) \quad [\text{C-DEFORD}] \\
&\text{if } (\text{dpv}(D) \cup \text{fpv}(D)) \cap \text{dpv}(D') = (\text{dpv}(D') \cup \text{fpv}(D')) \cap \text{dpv}(D) = \emptyset
\end{aligned}$$

## B Session Subtyping

We formalise our *subtyping* relation  $\leq$  in Def. 21 below. The relation is mostly standard [29, Def. 2.5], except for the new rule [SUB-stop], and the new (**highlighted**) side condition “ $|I| = 1 \implies \dots$ ” in rule [SUB- $\&$ ]: this condition prevents the supertype from adding input branches to “pure” crash recovery external choices.

► **Definition 21** (Subtyping). *Given a standard subtyping  $<$ : for basic types (e.g. including  $\text{int} <: \text{real}$ ), the session subtyping relation  $\leq$  is coinductively defined:*

$$\begin{aligned}
\frac{B <: B'}{B \leq B'} \quad [\text{SUB-B}] & \quad \frac{}{\text{end} \leq \text{end}} \quad [\text{SUB-end}] & \quad \frac{\forall i \in I \quad S'_i \leq S_i \quad T_i \leq T'_i}{\mathbf{p} \oplus \{m_i(S_i).T_i\}_{i \in I \cup J} \leq \mathbf{p} \oplus \{m_i(S'_i).T'_i\}_{i \in I}} \quad [\text{SUB-}\oplus] \\
\frac{}{\text{stop} \leq \text{stop}} \quad [\text{SUB-stop}] & \quad \frac{\forall i \in I \quad S_i \leq S'_i \quad T_i \leq T'_i \quad |I| = 1 \implies (m_i \neq \text{crash} \text{ or } J = \emptyset)}{\mathbf{p} \& \{m_i(S_i).T_i\}_{i \in I} \leq \mathbf{p} \& \{m_i(S'_i).T'_i\}_{i \in I \cup J}} \quad [\text{SUB-}\&] \\
\frac{T[\mu\mathbf{t}.T/\mathbf{t}] \leq T'}{\mu\mathbf{t}.T \leq T'} \quad [\text{SUB-}\mu\text{L}] & \quad \frac{T \leq T'[\mu\mathbf{t}.T'/\mathbf{t}]}{T \leq \mu\mathbf{t}.T'} \quad [\text{SUB-}\mu\text{R}]
\end{aligned}$$

Rule [SUB-B] lifts  $\leq$  to basic types. The rest of the rules say that a subtype describes a more permissive session protocol w.r.t. its supertype. By rule [SUB- $\oplus$ ], the subtype of an internal choice allows for selecting from a wider set of message labels, and sending more generic payloads. By rule [SUB- $\&$ ], the subtype of an external choice can support a smaller set of input message labels, and less generic payloads; the side condition “ $|I| = 1 \dots$ ” ensures that if the subtype only has a singleton **crash** branch, then the same applies to the supertype — hence, both subtype and supertype describe a “pure” crash recovery behaviour, and do not expect to receive any other input.<sup>3</sup> By rules [SUB-end] and [SUB-stop], the types **end** and **stop** are only subtypes of themselves. Finally, rules [SUB- $\mu$ L] and [SUB- $\mu$ R] say that recursive types are related up to their unfolding.

<sup>3</sup> Notice, however, that rule [SUB- $\&$ ] allows a supertype to have a **crash**-handling branch even when the subtype does not have one.



## C Additional Examples

► **Example 22.** We show an example of our crashing semantics. Processes  $P$  and  $Q$  below communicate on a session  $s$ ;  $P$  uses the endpoint  $s[\mathbf{p}]$  to send an endpoint  $s[\mathbf{r}]$  to role  $\mathbf{q}$ ;  $Q$  uses the endpoint  $s[\mathbf{q}]$  to receive an endpoint  $x$ , then sends a message to role  $\mathbf{p}$  via  $x$ .

$$P = s[\mathbf{p}][\mathbf{q}] \oplus m' \langle s[\mathbf{r}] \rangle . s[\mathbf{p}][\mathbf{r}] \& m(x) \quad Q = s[\mathbf{q}][\mathbf{p}] \& m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle$$

On a successful reduction (without crashes), we have:

$$\begin{aligned} & (\nu s) (P \mid Q) \\ &= (\nu s) (s[\mathbf{p}][\mathbf{q}] \oplus m' \langle s[\mathbf{r}] \rangle . s[\mathbf{p}][\mathbf{r}] \& m(x) \mid s[\mathbf{q}][\mathbf{p}] \& m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle) \\ &\rightarrow (\nu s) (s[\mathbf{p}][\mathbf{r}] \& m(x) \mid s[\mathbf{r}][\mathbf{p}] \oplus m \langle 42 \rangle) \\ &\rightarrow \mathbf{0} \end{aligned}$$

Now, suppose that  $P$  crashes before sending; this gives rise to the reduction:

$$\begin{aligned} & (\nu s) (P \mid Q) \\ &= (\nu s) (s[\mathbf{p}][\mathbf{q}] \oplus m' \langle s[\mathbf{r}] \rangle . s[\mathbf{p}][\mathbf{r}] \& m(x) \mid s[\mathbf{q}][\mathbf{p}] \& m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle) \\ &\rightarrow (\nu s) (s[\mathbf{p}] \dagger \mid s[\mathbf{r}] \dagger \mid s[\mathbf{q}][\mathbf{p}] \& m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle) \end{aligned}$$

We can observe that when the sending process  $P$  crashes (by  $[\mathbf{R}\text{-}\dagger \oplus]$ ), all endpoints in  $P$  (i.e. both  $s[\mathbf{p}]$  and  $s[\mathbf{r}]$ ) crash. If  $Q$  has a crash handling branch, it can be triggered via  $[\mathbf{R}\text{-}\odot]$ , suppose instead we have

$$Q' = s[\mathbf{q}][\mathbf{p}] \& \{m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle, \text{crash}.\mathbf{0}\}$$

A crash handling reduction can trigger when  $P$  crashes:

$$\begin{aligned} & (\nu s) (P \mid Q') \\ &= (\nu s) (s[\mathbf{p}][\mathbf{q}] \oplus m' \langle s[\mathbf{r}] \rangle . s[\mathbf{p}][\mathbf{r}] \& m(x) \mid s[\mathbf{q}][\mathbf{p}] \& \{m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle, \text{crash}.\mathbf{0}\}) \\ &\rightarrow (\nu s) (s[\mathbf{p}] \dagger \mid s[\mathbf{r}] \dagger \mid s[\mathbf{q}][\mathbf{p}] \& \{m'(x) . x[\mathbf{p}] \oplus m \langle 42 \rangle, \text{crash}.\mathbf{0}\}) \\ &\rightarrow (\nu s) (s[\mathbf{p}] \dagger \mid s[\mathbf{r}] \dagger \mid \mathbf{0}) \end{aligned}$$

► **Example 23.** Recall the types of the DNS example in § 1:

$$T'_p = \mathbf{q} \oplus \text{req} . \mathbf{q} \& \left\{ \begin{array}{l} \text{res} . \text{end} \\ \text{crash} . \mathbf{r} \oplus \text{req} . \mathbf{r} \& \text{res} . \text{end} \end{array} \right\} \quad \begin{array}{l} T'_q = \mathbf{p} \& \text{req} . \mathbf{p} \oplus \text{res} . \text{end} \\ T'_r = \mathbf{q} \& \text{crash} . \mathbf{p} \& \text{req} . \mathbf{p} \oplus \text{res} . \text{end} \end{array}$$

Now, consider the following typing context, containing such types:

$$\Gamma = s[\mathbf{p}]:T'_p, s[\mathbf{q}]:T'_q, s[\mathbf{r}]:T'_r$$

Such  $\Gamma$  is  $(s; \{\mathbf{p}, \mathbf{r}\})$ -safe. We can verify it by checking its reductions. When no crashes occur, we have the following two reductions, where each reductum satisfies Def. 8:

$$\begin{aligned} \Gamma &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\mathbf{q} \& \left\{ \begin{array}{l} \text{res} \\ \text{crash} . \mathbf{r} \oplus \text{req} . \mathbf{r} \& \text{res} \end{array} \right\}, s[\mathbf{q}]:\mathbf{p} \oplus \text{res}, s[\mathbf{r}]:T'_r \\ &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\text{end}, s[\mathbf{q}]:\text{end}, s[\mathbf{r}]:T'_r \end{aligned}$$

In the case where  $\mathbf{q}$  crashes immediately, we have:

$$\begin{aligned} \Gamma &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:T'_p, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:T'_r \\ &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\mathbf{q} \& \{\text{res}, \text{crash} . \mathbf{r} \oplus \text{req} . \mathbf{r} \& \text{res}\}, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:T'_r \\ &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\mathbf{r} \oplus \text{req} . \mathbf{r} \& \text{res}, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:T'_r \\ &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\mathbf{r} \oplus \text{req} . \mathbf{r} \& \text{res}, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:\mathbf{p} \& \text{req} . \mathbf{p} \oplus \text{res} \\ &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\mathbf{r} \& \text{res}, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:\mathbf{p} \oplus \text{res} \\ &\rightarrow_{\dagger \setminus s; \{\mathbf{p}, \mathbf{r}\}} s[\mathbf{p}]:\text{end}, s[\mathbf{q}]:\text{stop}, s[\mathbf{r}]:\text{end} \end{aligned}$$

and each reductum satisfies Def. 8. The case where  $\mathbf{q}$  crashes after receiving the **request** is similar. There are no other crash reductions to consider, since  $\mathbf{p}$  and  $\mathbf{r}$  are reliable.

► **Example 24.** We illustrate safety, deadlock-freedom, liveness, termination, and never-termination over typing contexts via a series of small examples. We first consider the typing context  $\Gamma_A = \Gamma_{Ap}, \Gamma_{Aq}, \Gamma_{Ar}$  where:

$$\begin{aligned}\Gamma_{Ap} &= s[p]:\mu t_p.q \oplus \{ok.q \& \{ok.t_p, ko.end, crash.end\}, ko.end\} \\ \Gamma_{Aq} &= s[q]:\mu t_q.p \& \{ok.p \oplus \{ok.t_q, ko.end\}, ko.end, crash.r \oplus ok.end\} \\ \Gamma_{Ar} &= s[r]:p \& \{crash.q \& \{ok.end, crash.end\}\}\end{aligned}$$

If we assume that all roles in  $\Gamma_A$  are unreliable,  $\Gamma_A$  is safe since its inputs/outputs are dual. However,  $\Gamma_A$  is *neither* deadlock-free *nor* live since it is possible for  $p$  to crash immediately before  $q$  sends  $ko$  to  $p$ . In such cases,  $q$  will *not* detect that  $p$  has crashed (since we only detect crashes on receive actions) and terminate *without* sending a message to the backup process  $r$ . This results in a deadlock because  $r$  *will* detect that  $p$  has crashed, and *will* expect a message from  $q$ .

We observe that changing the reliability assumptions, without changing the typing context, may influence whether a typing context property holds. For example, consider the typing context  $\Gamma_B = \Gamma_{Bp}, \Gamma_{Bq}, \Gamma_{Br}$  where:

$$\begin{aligned}\Gamma_{Bp} &= s[p]:\mu t_p.q \oplus ok.t_p \\ \Gamma_{Bq} &= s[q]:\mu t_q.p \& \{ok.t_q, crash.\mu t'_q.r \& \{ok.t'_q, crash.end\}\} \\ \Gamma_{Br} &= s[r]:\mu t_r.q \oplus ok.t_r\end{aligned}$$

If we assume that all roles are unreliable,  $\Gamma_B$  is safe and deadlock-free but *not* live — because  $p$  may never crash, and in this case,  $r$ 's outputs are never received by  $q$ . Notably,  $\Gamma_B$  is *not* never-terminating because if both  $p$  and  $r$  crash, then the surviving  $q$  can reach  $end$ ; however, if we assume that just  $r$  is reliable (i.e.  $\mathcal{R} = \{r\}$ ), then  $\Gamma_B$  becomes also never-terminating — because even if both  $p$  and  $q$  crash, role  $r$  can keep running by sending forever  $ok$  messages that are lost (by rule  $[r\text{-}l_m]$  in Fig. 3).

Notice that, in the case of  $\Gamma_B$ , we are unable to make liveness hold purely via combinations of reliable roles: this is because (unless  $p$  crashes)  $r$ 's output will never be received by  $q$ , irrespective of reliability assumptions. The typing context itself must instead be adapted; for example, by only permitting  $r$  to send once it has detected that  $p$  has crashed.

Instead, in the case of  $\Gamma_A$ , we *can* obtain liveness by adjusting the reliability assumptions: in fact, if we assume  $r \in \mathcal{R}$ , then  $\Gamma_A$  is both deadlock-free and live.

Finally, consider the typing context  $\Gamma_C = \Gamma_{Cp}, \Gamma_{Cq}, \Gamma_{Cr}$  where:

$$\begin{aligned}\Gamma_{Cp} &= s[p]:q \oplus m_1.q \& \{m_2.end, crash.\mu t_p.r \oplus ok.t_p\} \\ \Gamma_{Cq} &= s[q]:p \& \{m_1.p \oplus m_2.end\} \\ \Gamma_{Cr} &= s[r]:p \& \{crash.\mu t_q.p \& \{ok.t_q\}\}\end{aligned}$$

$\Gamma_C$  satisfies safety, deadlock-freedom, and termination when *all* roles are assumed to be reliable. However, should we instead assume that only  $p$  is reliable, then  $\Gamma_C$  does not satisfy termination. Since external choices in  $\Gamma_C$  do not feature a crash-handling branch when receiving from  $p$ , should no roles be assumed reliable,  $\Gamma_C$  satisfies only safety.

## D Tool Evaluation

To verify the properties in Fig. 5, we extend the Multiparty Session Types toolKit (mpstk) [30], which uses the mCRL2 model checker [4]. Our extended tool is available at:

<https://github.com/alcestes/mpstk-crash-stop>

We evaluate our approach with 5 examples: DNS, from § 1; Adder, TwoBuyers, and Negotiate, extended from the session type literature [35] with crashes and crash handling

|                   |  |
|-------------------|--|
| ( $\alpha$ )      | $s[p]:q \oplus req.q \& \{res.end, crash.r \oplus req.r \& res.end\}$ $s[q]:p \& req.p \oplus res.end$ $s[r]:q \& crash.p \& req.p \oplus res.end$   |
| ( $\beta$ )       | $s[p]:\mu t.q \oplus add(Int).q \oplus \{add(Int).q \& \{res(Int).t, T_i\}, ko.q \& \{T_{ko}, T_i\}\}$ $s[q]:\mu t.p \& \{add(Int).p \& \{add(Int).p \oplus res(Int).t, ko.p \oplus T_{ko}, T_i\}, T_i\}$  |
| ( $\gamma$ )      | $s[b1]:s \oplus r(Str).s \& \{q(Int).b2 \oplus s(Int).b2 \& \{crash.T_1\}, crash.b2 \oplus T_{ko}\}$ $T_1 = s \& \{rp1.s \oplus \{ok.T_2, T_{ko}\}, rp2.T_2, rp3.s \& \{d(Str).end, T_i\}, T_i\}$ $T_2 = s \oplus a(Str).s \& \{d(Str).end, T_i\}$ $s[b2]:s \& \{q(Int).T_1, T_{ko}, crash.T_1\}$ $T_1 = b1 \& \{s(Int).s \oplus \{ok.s \oplus a(Str).s \& \{d(Str).end, T_i\}, T_{ko}\}, T_{ko}, crash.s \oplus T_{ko}\}$ $s[s]:b1 \& \{r(Str).b1 \oplus q(Int).b2 \oplus q(Int).b2 \& \{ok.T_1, T_{ko}, crash.b1 \oplus rp1.T_2\}, crash.b2 \oplus T_{ko}\}$ $T_1 = b2 \& \{a(Str).b2 \oplus d(Str).b2 \& \{crash.b1 \oplus rp3.T_4\}, crash.b1 \oplus rp2.T_3\}$ $T_2 = b1 \& \{ok.T_3, T_{ko}, T_i\} \quad T_3 = b1 \& \{a(Str).T_4, T_i\} \quad T_4 = b1 \oplus d(Str).end$ |
| ( $\delta$ )      | $s[n]:c \& \{o(Int).\mu t.c \oplus g.c \oplus \{o(Int).c \& \{o(Int).t, ok.c \oplus T_{ok}, T_{ko}, T_i\}, ok.c \& \{T_{ok}, T_i\}, T_{ko}\}, T_i\}$ $s[c]:n \oplus o(Int).\mu t.o.n \& \{g.n \& \{o(Int).T_1, ok.n \& \{crash.b \oplus T_{ok}\}, T_{ko}, crash.T_2\}, crash.T_2\}$ $T_1 = n \oplus \{o(Int).t.o, ok.n \& \{T_{ok}, crash.T_2\}, ko.n \& \{crash.b \oplus T_{ko}\}\}$ $T_2 = b \oplus o(Int).\mu t1.b \& \{g.b \& \{o(Int).b \oplus \{o(Int).t1, ok.b \& \{T_{ok}\}, T_{ko}\}, ok.b \& \{T_{ok}\}, T_{ko}\}\}$ $s[b]:n \& \{crash.c \& \{o(Int).\mu t.c \oplus g.c \oplus \{o(Int).T_1, ok.T_2, T_{ko}\}, T_{ok}, T_{ko}, T_i\}\}$ $T_1 = c \& \{o(Int).t, ok.c \oplus T_{ok}, T_{ko}\} \quad T_2 = c \& \{ok.c \oplus T_{ok}, T_i\}$                            |
| ( $\varepsilon$ ) | $s[p]:q \oplus data(Str).r \oplus data(Str).end$ $s[q]:p \& \{data(Str).p \& \{crash.r \& \{h.r \oplus data(Str).end, T_i\}\}, crash.r \& \{req.r \oplus T_{ko}, T_i\}\}$ $s[r]:p \& \{data(Str).end, crash.q \oplus req.q \& \{data(Str).end, T_{ko}, T_i\}\}$  |

■ **Figure 7** Typing contexts for ( $\alpha$ ) DNS, ( $\beta$ ) Adder, ( $\gamma$ ) TwoBuyers, ( $\delta$ ) Negotiate, and ( $\varepsilon$ ) Broadcast. Roles **p** and **q** of DNS, and **b** of Negotiate are reliable; all other roles are unreliable. Let  $T_i = crash.end$ ,  $T_{ko} = ko.end$ , and  $T_{ok} = ok.end$ .

behaviour; and Broadcast, inspired by the reliable broadcast algorithms in [5, Ch. 3]. The full typing contexts for each example are given in Fig. 7. We model and verify both fully reliable and (partially) unreliable versions of each example. In all examples, we show how the introduction of unreliability leads to an increase of model sizes and verification times. The increased model size reflects how the addition of crash handling can complicate even simple protocols, and motivates the use of automatic model checking. Still, we show that the verification of our examples always completes in less than 100 ms.

## D.1 Description of the Examples in Fig. 7

**DNS** is the example described in §1. The example demonstrates both backup processes and optional reliability assumptions.

**Adder** demonstrates a minimal extension of the fully reliable protocol, in which **q** receives two numbers from **p**, sums them, and communicates the result to **p**. In our extension, both roles are unreliable and the protocol ends when a crash is detected. It satisfies safety, deadlock-freedom, and liveness.

**TwoBuyers** is the example described in §4. It assumes that both the seller and buyers **b1** and **b2** are unreliable. In cases where the split has been agreed upon, and **b2** has crashed, **b1** concludes the sale. It satisfies safety, deadlock-freedom, liveness, and terminating. This form of TwoBuyers is not projectable from a global type, since **b1** would need to be informed on conclusion of a sale. TwoBuyers uses recovery behaviour in order to satisfy deadlock-freedom. Finally, TwoBuyers demonstrates the flexibility of crash-handling that our approach permits: **b2** does not alter its behaviour having detected that **s** has crashed (i.e. continues as  $T_1$ ), instead leaving **b1** to instigate crash-handling behaviour.

|                 | $\mathcal{R}$  | states | transitions | safe            | df              | live            | nterm           | term            |
|-----------------|----------------|--------|-------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| $(\alpha)$      | $\{p, r\}$     | 101    | 427         | $12.28 \pm 1\%$ | $17.14 \pm 1\%$ | $11.24 \pm 1\%$ | $15.47 \pm 0\%$ | $12.33 \pm 0\%$ |
|                 | $\mathfrak{R}$ | 10     | 15          | $7.61 \pm 1\%$  | $8.23 \pm 1\%$  | $7.46 \pm 1\%$  | $7.78 \pm 1\%$  | $7.6 \pm 1\%$   |
| $(\beta)$       | $\emptyset$    | 37     | 159         | $12.43 \pm 0\%$ | $15.74 \pm 0\%$ | $12.24 \pm 1\%$ | $14.46 \pm 0\%$ | $12.06 \pm 1\%$ |
|                 | $\mathfrak{R}$ | 26     | 56          | $8.92 \pm 2\%$  | $10.06 \pm 0\%$ | $8.71 \pm 1\%$  | $9.42 \pm 0\%$  | $8.79 \pm 0\%$  |
| $(\gamma)$      | $\emptyset$    | 1409   | 10248       | $45.6 \pm 0\%$  | $88.26 \pm 0\%$ | $31.33 \pm 0\%$ | $77.2 \pm 0\%$  | $45.65 \pm 0\%$ |
|                 | $\mathfrak{R}$ | 169    | 510         | $11.12 \pm 1\%$ | $15.94 \pm 0\%$ | $10.9 \pm 1\%$  | $12.19 \pm 0\%$ | $11.06 \pm 1\%$ |
| $(\delta)$      | $\{b\}$        | 1089   | 8106        | $34.61 \pm 0\%$ | $55.07 \pm 0\%$ | $25.69 \pm 0\%$ | $47.46 \pm 0\%$ | $26.04 \pm 0\%$ |
|                 | $\mathfrak{R}$ | 50     | 157         | $10.17 \pm 0\%$ | $12.7 \pm 0\%$  | $9.93 \pm 0\%$  | $11.33 \pm 0\%$ | $9.72 \pm 0\%$  |
| $(\varepsilon)$ | $\emptyset$    | 161    | 925         | $17.99 \pm 1\%$ | $28.13 \pm 0\%$ | $14.08 \pm 0\%$ | $25.72 \pm 1\%$ | $17.74 \pm 0\%$ |
|                 | $\mathfrak{R}$ | 13     | 25          | $7.85 \pm 3\%$  | $8.65 \pm 1\%$  | $7.7 \pm 0\%$   | $8.12 \pm 1\%$  | $7.85 \pm 0\%$  |

■ **Table 1** Average times (in milliseconds  $\pm$  std. dev.) for the verification of DNS ( $\alpha$ ), Adder ( $\beta$ ), TwoBuyers ( $\gamma$ ), Negotiate ( $\delta$ ), and Broadcast ( $\varepsilon$ ) in Fig. 7 over safety (safe), deadlock-freedom (df), liveness (live), never-terminating (nterm) and terminating (term). Each example has two rows of measurements, varying the sets of reliable roles  $\mathcal{R}$ : either zero/one/two reliable roles (first row), or all reliable roles (second row). (Benchmarking specs: Intel Core i7-7700K CPU, 4.20 GHz, 16 GB RAM, mCRL2 202106.0 invoked 30 times with: `pbcs2bool --solve-strategy=2.`)

**Negotiate** introduces a (reliable) **backup** negotiator to the version found in the literature.

During normal operation, a **client** will send an opening offer to a **negotiator**. Both **c** and **n** can then choose to repeatedly exchange counter offers until the other accepts the offer, or rejects it outright, bringing the protocol to an end. In our extension, should the **customer** detect that the original **negotiator** crashes, the **backup** negotiator activates and continues the negotiation with **c**. The example satisfies safety, deadlock-freedom, and liveness. Recovery actions are necessary for **c** in two locations in order to avoid deadlocks: it is otherwise possible for an **offer** to be declined or agreed upon, then for **n** to crash without **c** noticing; this results in **b** activating, and expecting a message from the terminated **c**.

**Broadcast** contains an unreliable broadcaster **p** attempting to send **data** to two receivers **q** and **r**. In cases where **p** crashes, **r** requests the data from **q**, who responds with the data it received before **p** crashed, or with **ko** when **p** crashed immediately. The example is not projectable from a global type, since **q** would otherwise require a message from **r** even when **p** had not crashed. **Broadcast** satisfies safety, deadlock-freedom, liveness, and termination. As in **Negotiate**, recovery behaviour is necessary for **Broadcast** to satisfy deadlock-freedom.

Notably, **Adder**, **TwoBuyers** and **Broadcast** have *no* reliability assumptions: any role may crash at any point. Barring **Adder**, our examples cannot be written using *global types* in the session types literature. This demonstrates the flexibility of our generalised MPST system over the classic one. Moreover, the examples include the use of failover processes (**DNS** and **Negotiate**) and complex recovery behaviour (**TwoBuyers**, **Negotiate**, and **Broadcast**), thus showcasing the expressivity of our approach.

## D.2 Experimental Results

We applied our extended implementation of `mpstk` to the examples in Fig. 7. Table 1 gives the full set of verification times, reported in milliseconds with standard deviations, where each time is an average of 30 runs.

For each example, we give verification times for both the typing contexts in Fig. 7 and a corresponding fully reliable version (i.e. where all roles in the protocol are reliable;  $\mathcal{R} = \mathfrak{R}$ ). For `Adder`, `TwoBuyers`, and `Negotiate`, we use the standard protocol definitions from the literature. For `DNS` and `Broadcast`, we omit crash-handling branches. For `DNS`, this has the consequence of removing the backup role `r` entirely.

All examples satisfy safety, deadlock-freedom, and liveness; `Adder` and `Broadcast` satisfy termination; no example satisfies never-termination.

Unsurprisingly, all examples demonstrate an increase in verification times and the number of states and transitions when comparing unreliable to reliable versions. Even `Adder`, which represents minimal crash-handling, demonstrates relevant increases to the number of states and transitions: this is a direct consequence of the unreliable roles, and the resulting generation of crash and crash-detection transitions in the LTS generated by mCRL2. Verification times also increase because the verified properties follow crash and communication actions, thus requiring the exploration of a larger state space compared to the fully reliable versions.

Nevertheless, our verification times do not increase as quickly as the state space grows, and are always under 100 ms. This is because our  $\mu$ -calculus formulæ only follow communication, crash, and crash detection transitions, and thus, their verification may not need to follow every possible transition into every state. This suggests greater scalability of the approach that would otherwise be suggested by the size of the state space. This also lends greater motivation to the use of model checkers, as it is infeasible to manually determine the properties of a large LTS with complex crash-handling behaviour.