

Designing Asynchronous Multiparty Protocols with Crash-Stop Failures

Adam D. Barwell  



University of St. Andrews, UK
University of Oxford, UK

Ping Hou  

University of Oxford, UK

Nobuko Yoshida  

University of Oxford, UK

Fangyi Zhou  

Imperial College London, UK
University of Oxford, UK

Abstract

Session types provide a typing discipline for message-passing systems. However, most session type approaches assume an ideal world: one in which everything is reliable and without failures. Yet this is in stark contrast with distributed systems in the real world. To address this limitation, we introduce TEATRINO, a code generation toolchain that utilises asynchronous *multiparty session types* (MPST) with *crash-stop* semantics to support failure handling protocols.

We augment asynchronous MPST and processes with *crash handling* branches. Our approach requires no user-level syntax extensions for global types and features a formalisation of global semantics, which captures complex behaviours induced by crashed/crash handling processes. The sound and complete correspondence between global and local type semantics guarantees deadlock-freedom, protocol conformance, and liveness of typed processes in the presence of crashes.

Our theory is implemented in the toolchain TEATRINO, which provides *correctness by construction*. TEATRINO extends the SCRIBBLE multiparty protocol language to generate protocol-conforming SCALA code, using the EFFPI concurrent programming library. We extend both SCRIBBLE and EFFPI to support *crash-stop* behaviour. We demonstrate the feasibility of our methodology and evaluate TEATRINO with examples extended from both session type and distributed systems literature.

2012 ACM Subject Classification Software and its engineering → Source code generation; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi; Theory of computation → Distributed computing models

Keywords and phrases Session Types, Concurrency, Failure Handling, Code Generation, Scala

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.1

Related Version *Full Version*: <https://arxiv.org/abs/2305.06238>

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*:

<https://doi.org/10.4230/DARTS.9.2.9>

Software (ECOOP 2023 Artifact Evaluation approved artifact):

<http://doi.org/10.5281/zenodo.7714133>

Software (Source Code): <https://github.com/adbarwell/EC00P23-Artefact>

archived at `swh:1:dir:e680ab478b62aab45610b0ef9f6de9d0f9be20ad2`

Funding Work supported by: EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We thank the anonymous reviewers for their useful comments and suggestions. We thank Jia Qing Lim for his contribution to the EFFPI extension. We thank Alceste Scalas for useful discussions and advice in the development of this paper and for his assistance with EFFPI.



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

37th European Conference on Object-Oriented Programming (ECOOP 2023).

Editors: Karim Ali and Guido Salvaneschi; Article No. 1; pp. 1:1–1:30



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



1 Introduction

Background. As distributed programming grows increasingly prevalent, significant research effort has been devoted to improve the reliability of distributed systems. A key aspect of this research focuses on studying *unreliability* (or, more specifically, failures). Modelling unreliability and failures enables a distributed system to be designed to be more tolerant of failures, and thus more resilient.

In pursuit of methods to achieve safety in distributed communication systems, *session types* [19] provide a lightweight, type system-based approach to message-passing concurrency. In particular, *Multiparty Session Types* (MPST) [20] facilitate the specification and verification of communication between message-passing processes in concurrent and distributed systems. The typing discipline prevents common communication-based errors, e.g. deadlocks and communication mismatches [21, 37]. On the practical side, MPST have been implemented in various mainstream programming languages [7, 10, 11, 22, 24, 25, 28, 30], which facilitates their applications in real-world programs.

Nevertheless, the challenge to account for unreliability and failures persists for session types: most session type systems assume that both participants and message transmissions are *reliable* without failures. In a real-world setting, however, participants may crash, communications channels may fail, and messages may be lost. The lack of failure modelling in session type theories creates a barrier to their applications to large-scale distributed systems.

Recent works [3, 26, 27, 33, 42] close the gap of failure modelling in session types with various techniques. [42] introduces *failure suspicion*, where a participant may suspect their communication partner has failed, and act accordingly. [33] introduces *reliability annotations* at type level, and fall back to a given *default* value in case of failures. [26] proposes a framework of *affine* multiparty session types, where a session can terminate prematurely, e.g. in case of failures. [3] integrates *crash-stop failures*, where a generalised type system validates safety and liveness properties with model checking; [27] takes a similar approach, modelling more kinds of failures in a session type system, e.g. message losses, reordering, and delays.

While steady advancements are made on the theoretical side, the implementations of those enhanced session type theories seem to lag behind. Barring the approaches in [26, 42], the aforementioned approaches [3, 27, 33] do not provide session type API support for programming languages.¹ To bring the benefits of the theoretical developments into real-world distributed programming, a gap remains to be filled on the implementation side.

This Paper. We introduce a *top-down* methodology for designing asynchronous multiparty protocols with crash-stop failures:

- (1) We use an extended asynchronous MPST theory, which models *crash-stop* failures, and show that the usual session type guarantees remain valid, i.e. communication safety, deadlock-freedom, and liveness;
- (2) We present a toolchain for implementing asynchronous multiparty protocols, under our new asynchronous MPST theory, in SCALA, using the EFFPI concurrency library [38].

The top-down design methodology comes from the original MPST theory [20], where the design of multiparty protocols begins with a given *global* type (top), and implementations rely on *local* types (bottom) obtained from the global type. The global and local types reflect the

¹ [3] provides a prototype implementation, utilising the mCRL2 model checker [5], for verifying type-level properties, instead of a library for general use.

global and local communication behaviours respectively. Well-typed implementations that conform to a global type are guaranteed to be *correct by construction*, enjoying full guarantees (safety, deadlock-freedom, liveness) from the theory. This remains the predominant approach for implementing MPST theories, and is also followed by some aforementioned systems [26, 42].

We model *crash-stop* failures [6, §2.2], i.e. a process may fail arbitrarily and cease to interact with others. This model is simple and expressive, and has been adopted by other approaches [3, 27]. Using global types in our design for handling failures in multiparty protocols presents two distinct advantages:

- (1) global types provide a simple, high-level means to both specify a protocol abstractly and automatically derive local types; and,
- (2) desirable behavioural properties such as communication safety, deadlock-freedom, and liveness are guaranteed by construction.

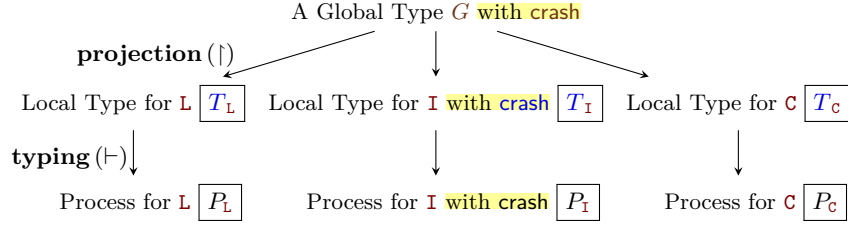
In contrast to the *synchronous* semantics in [3], we model an *asynchronous* semantics, where messages are buffered whilst in transit. We focus on asynchronous systems since most communication in the real distributed world is asynchronous. In [27], although the authors develop a generic typing system incorporating asynchronous semantics, their approach results in the type-level properties becoming undecidable [27, §4.4]. With global types, we restore the decidability at a minor cost to expressivity.

To address the gap on the practical side, we present a code generator toolchain, TEATRINO, to implement our MPST theory. Our toolchain takes an asynchronous multiparty protocol as input, using the protocol description language SCRIBBLE [43], and generates SCALA code using the EFFPI [38] concurrency library as output.

The SCRIBBLE Language [43] is designed for describing multiparty communication protocols, and is closely connected to MPST theory (cf. [31]). This language enables a programmatic approach for expressing global types and designing multiparty protocols. The EFFPI concurrency library [38] offers an embedded Domain Specific Language (DSL) that provides a simple actor-based API. The library offers both type-level and value-level constructs for processes and channels. Notably, the type-level constructs reflect the behaviour of programs (i.e. processes) and can be used as specifications. Our code generation technique, as well as the EFFPI library itself, utilises the type system features introduced in SCALA 3, including match types and dependent function types, to encode local types in EFFPI. This approach enables us to specify and verify program behaviour at the type level, resulting in a more powerful and flexible method for handling concurrency.

By extending SCRIBBLE and EFFPI to support *crash detection and handling*, our toolchain TEATRINO provides a lightweight way for developers to take advantage of our theory, bridging the gap on the practical side. We evaluate the expressivity and feasibility of TEATRINO with examples incorporating crash handling behaviour, extended from session type literature.

Outline. We begin with an overview of our methodology in § 2. We introduce an asynchronous multiparty session calculus in § 3 with semantics of crashing and crash handling. We introduce an extended theory of asynchronous multiparty session types with semantic modelling of crash-stop failures in § 4. We present a typing system for the multiparty session calculus in § 5. We introduce TEATRINO, a code generation toolchain that implements our theory in § 6, demonstrating how our approach is applied in the SCALA programming language. We evaluate our toolchain with examples from both session type and distributed systems literature in § 7. We discuss related work in § 8 and conclude in § 9. Full proofs, auxiliary material, and more details of TEATRINO can be found in the full version of the paper [2]. Additionally, our toolchain and examples used in our evaluation are available on [GitHub](#).



■ **Figure 1** Top-down View of MPST with Crash.

2 Overview

In this section, we give an overview of our methodology for designing asynchronous multiparty protocols with crash-stop failures, and demonstrate our code generation toolchain, TEATRINO.

Asynchronous Multiparty Protocols with Crash-Stop Failures. We follow a standard top-down design approach enabling *correctness by construction*, but enrich asynchronous MPST with crash-stop semantics. As depicted in Fig. 1, we formalise (asynchronous) multiparty protocols with crash-stop failures as global types with *crash handling branches* (*crash*). These are projected into local types, which may similarly contain crash handling branches (*crash*). The projected local types are then used to type-check processes (also with crash handling branches (*crash*)) that are written in a session calculus. As an example, we consider a simple *distributed logging* scenario, which is inspired by the logging-management protocol [26], but extended with a third participant.

The Simpler Logging protocol consists of a *logger* (**L**) that controls the logging services, an *interface* (**I**) that provides communications between logger and client, and a *client* (**C**) that requires logging services via interface. Initially, **L** sends a heartbeat message *trigger* to **I**. Then **C** sends a command to **L** to read the logs (*read*). When a *read* request is sent, it is forwarded to **L**, and **L** responds with a *report*, which is then forwarded onto **C**. Assuming all participants (logger, interface, and client) are reliable, i.e. without any failures or crashes, this logging behaviour can be represented by the *global type* G_0 :

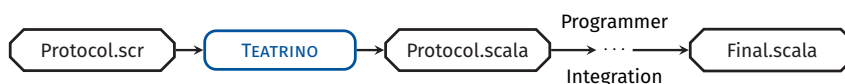
$$G_0 = \mathbf{L} \rightarrow \mathbf{I} : \text{trigger} . \mathbf{C} \rightarrow \mathbf{I} : \text{read} . \mathbf{I} \rightarrow \mathbf{L} : \text{read} . \mathbf{L} \rightarrow \mathbf{I} : \text{report}(\text{log}) . \mathbf{I} \rightarrow \mathbf{C} : \text{report}(\text{log}) . \text{end} \quad (1)$$

Here, G_0 is a specification of the Simpler Logging protocol between multiple roles from a global perspective.

In the real distributed world, all participants in the Simpler Logging system may fail. Ergo, we need to model protocols with failures or crashes and handling behaviour, e.g. should the client fail after the logging has started, the interface will inform the logger to stop and exit. We follow [6, §2.2] to model a *crash-stop* semantics, where we assume that roles can crash *at any time* unless assumed *reliable* (never crash). For simplicity, we assume **I** and **L** to be reliable. The above logging behaviour, incorporating crash-stop failures, can be represented by extending G_0 with a branch handling a crash of **C**:

$$G = \mathbf{L} \rightarrow \mathbf{I} : \text{trigger} . \mathbf{C} \rightarrow \mathbf{I} : \left\{ \begin{array}{l} \text{read} . \mathbf{I} \rightarrow \mathbf{L} : \text{read} . \mathbf{L} \rightarrow \mathbf{I} : \text{report}(\text{log}) . \mathbf{I} \rightarrow \mathbf{C} : \text{report}(\text{log}) . \text{end} \\ \text{crash} . \mathbf{I} \rightarrow \mathbf{L} : \text{fatal} . \text{end} \end{array} \right\} \quad (2)$$

We model crash detection on receiving roles: when **I** is waiting to receive a message from **C**, the receiving role **I** is able to detect whether **C** has crashed. Since crashes are detected only by the receiving role, we do not require a crash handling branch on the communication step between **I** and **C** – nor do we require them on any interaction between **L** and **I** (since we are assuming that **L** and **I** are reliable).



■ **Figure 2** Workflow of TEATRINO.

Following the MPST top-down methodology, a global type is then *projected* onto *local types*, which describe communications from the perspective of a single role. In our unreliable Simpler Logging example, G is projected onto three local types (one for each role C, L, I):

$$T_C = I \oplus \text{read}.I \& \text{report}(\text{log}).\text{end} \quad T_L = I \oplus \text{trigger}.I \& \left\{ \begin{array}{l} \text{read}.I \oplus \text{report}(\text{log}).\text{end} \\ \text{fatal}.\text{end} \end{array} \right\}$$

$$T_I = L \& \text{trigger}.C \& \left\{ \begin{array}{l} \text{read}.L \oplus \text{read}.L \& \text{report}(\text{log}).C \oplus \text{report}(\text{log}).\text{end} \\ \text{crash}.L \oplus \text{fatal}.\text{end} \end{array} \right\}$$

Here, T_I states that I first receives a trigger message from L ; then I either expects a **read** request from C , or detects the crash of C and handles it (in **crash**) by sending the **fatal** message to notify L . We add additional crash modelling and introduce a **stop** type for crashed endpoints. We show the operational correspondence between global and local type semantics, and demonstrate that a projectable global type always produces a safe, deadlock-free, and live typing context.

The next step in this top-down methodology is to use local types to type-check processes P_i executed by role p_i in our session calculus. For example, T_I can be used to type check I that executes the process:

$$L? \text{trigger} . \sum \left\{ \begin{array}{l} C? \text{read}.L! \text{read}.L? \text{report}(x).C! \text{report}(x).0 \\ C? \text{crash}.L! \text{fatal}.0 \end{array} \right\}$$

In our operational semantics (§3), we allow active processes executed by unreliable roles to crash *arbitrarily*. Therefore, the role executing the crashed process also crashes, and is assigned the local type **stop**. To ensure that a communicating process is type-safe even in presence of crashes, we require that its typing context satisfies a *safety property* accounting for possible crashes (Def. 13), which is preserved by projection. Additional semantics surrounding crashes adds subtleties even in standard results. We prove subject reduction and session fidelity results accounting for crashes and sets of reliable roles.

Code Generation Toolchain: Teatrino. To complement the theory, we present a code generation toolchain, TEATRINO, that generates protocol-conforming SCALA code from a multiparty protocol. We show the workflow diagram of our toolchain in Fig. 2. TEATRINO takes a SCRIBBLE protocol (Protocol.scr) and generates executable code (Protocol.scala) conforming to that protocol, which the programmer can integrate with existing code (Final.scala).

TEATRINO implements our session type theory to handle global types expressed using the SCRIBBLE protocol description language [43], a programmer-friendly way for describing multiparty protocols. We extend the syntax of SCRIBBLE slightly to include constructs for **crash** recovery branches and reliable roles.

The generated SCALA code utilises the EFFPI concurrency library [38]. EFFPI is an embedded domain specific language in SCALA 3 that offers a simple Actor-based API for concurrency. Our code generation technique, as well the EFFPI library itself, leverages the type system features introduced in SCALA 3, e.g. match types and dependent function types, to encode local types in EFFPI. We extend EFFPI to support crash detection and handling.

As a brief introduction to EFFPI, the concurrency library provides types for processes and channels. For processes, an output process type `Out[A, B]` describes a process that uses a channel of type `A` to send a value of type `B`, and an input process type `In[A, B, C]` describes a process that uses a channel of type `A` to receive a value of type `B`, and pass it to a continuation type `C`. Process types can be sequentially composed by the `>>>` operator. For channels, `Chan[X]` describes a channel that can be used communicate values of type `X`. More specifically, the usage of a channel can be reflected at the type level, using the types `InChan[X]/OutChan[X]` for input/output channels.

```

1 type I[C0 <: InChan[Trigger], C1 <: OutChan[Fatal],
2     C2 <: InChan[Read], C3 <: InChan[Report], C4 <: OutChan[Report]]
3 = InErr[C0, Trigger,
4     (X <: Read) =>
5     Out[C3,Read] >>> In[C4, Report, (Y <: Log) => Out[C5, Report]],
6     (Err <: Throwable) => Out[C2,Fatal]
7 ]

```

■ **Figure 3** EFFPI Type for T_I .

As a sneak peek of the code we generate, in Fig. 3, we show the generated EFFPI representation for the projected local type T_I from the Simpler Logging example. Readers may be surprised by the difference between T_I and the generated EFFPI type `I`. This is because the process types need their respective channel types, namely the type variables `C0`, `C1`, *etc.* bounded by `InChan[...]` and `OutChan[...]`. We explain the details of code generation in §6.2, and describe an interesting challenge posed by the channel generation procedure in §6.3.

For crash handling behaviour, we introduce a new type `InErr`, whose last argument specifies a continuation type to follow in case of a crash. Line 3 in Fig. 3 shows the crash handling behaviour: sending a message of type `Fatal`, which reflects the `crash` branch in the local type T_I . We give more details of the generated code in §6.2.

Code generated by TEATRINO is executable, protocol-conforming, and can be specialised by the programmer to integrate with existing code. We evaluate our toolchain on examples taken from both MPST and distributed programming literature in §7. Moreover, we extend each example with crash handling behaviour to define unreliable variants. We demonstrate that, with TEATRINO, code generation takes negligible time, and all potential crashes are accompanied with crash handlers.

3 Crash-Stop Asynchronous Multiparty Session Calculus

In this section, we formalise the syntax and operational semantics of our asynchronous multiparty session calculus with process failures and crash detection.

Syntax. Our asynchronous multiparty session calculus models processes that may crash arbitrarily. Our formalisation is based on [16] – but in addition, follows the *fail-stop* model in [6, §2.7], where processes may crash and never recover, and process failures can be detected by failure detectors [6, §2.6.2] [8] when attempting to receive messages.

We give the syntax of processes in Fig. 4. In our calculus, we assume that there are basic expressions e (e.g. `true`, `false`, `7 + 11`) that are assigned basic types B (e.g. `int`, `bool`). We write $e \downarrow v$ to denote an expression e evaluates to a value v (e.g. $(7 < 11) \downarrow \text{true}$, $(1 + 1) \downarrow 2$).

$P, Q ::=$ $\sum_{i \in I} \mathbf{p}^?m_i(x_i).P_i$ $\mathbf{p}!m(e).P \quad (\text{where } m \neq \text{crash})$ $\text{if } e \text{ then } P \text{ else } Q$ X $\mu X.P$ $\mathbf{0}$ ζ	Processes <i>external choice</i> <i>output</i> <i>conditional</i> <i>variable</i> <i>recursion</i> <i>inaction</i> <i>crashed</i>	$\mathcal{M} ::=$ $\mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h$ $\mathcal{M} \mid \mathcal{M}$ $h ::=$ ϵ \emptyset $(\mathbf{p}, m(v))$ $h \cdot h$	Sessions <i>participant</i> <i>parallel</i> Queues <i>empty</i> <i>unavailable</i> <i>message</i> <i>concatenation</i>
--	---	---	---

■ **Figure 4** Syntax of sessions, processes, and queues. Noticeable changes w.r.t. standard session calculus [16] are highlighted.

A process, ranged over by P, Q , is a communication agent within a session. An *output* process $\mathbf{p}!m(e).P$ sends a message to another role \mathbf{p} in the session, where the message is labelled m , and carries a payload expression e , then the process continues as P . An *external choice (input)* process $\sum_{i \in I} \mathbf{p}^?m_i(x_i).P_i$ receives a message from another role \mathbf{p} in the session, among a finite set of indexes I , if the message is labelled m_i , then the payload would be received as x_i , and process continues as P_i . Note that our calculus uses *crash* as a special message label denoting that a participant (role) has crashed. Such a label cannot be sent by any process, but a process can implement crash detection and handling by receiving it. Consequently, an output process *cannot* send a *crash* message (side condition $m \neq \text{crash}$), whereas an input process may include a *crash handling branch* of the form $\text{crash}.P'$ meaning that P' is executed when the sending role has crashed. A *conditional* process $\text{if } e \text{ then } P \text{ else } Q$ continues as either P or Q depending on the evaluation of e . We allow *recursion* at the process level using $\mu X.P$ and X , and we require process recursion variables to be guarded by an input or an output action; we consider a recursion process structurally congruent to its unfolding $\mu X.P \equiv P\{\mu X.P/X\}$. Finally, we write $\mathbf{0}$ for an *inactive* process, representing a successful termination; and ζ for a *crashed* process, representing a termination due to failure.

An *incoming queue*², ranged over by h, h' , is a sequence of messages tagged with their origin. We write ϵ for an *empty* queue; \emptyset for an *unavailable* queue; and $(\mathbf{p}, m(v))$ for a message sent from \mathbf{p} , labelled m , and containing a payload value v . We write $h_1 \cdot h_2$ to denote the concatenation of two queues h_1 and h_2 . When describing incoming queues, we consider two messages from different origins as swappable: $h_1 \cdot (\mathbf{q}_1, m_1(v_1)) \cdot (\mathbf{q}_2, m_2(v_2)) \cdot h_2 \equiv h_1 \cdot (\mathbf{q}_2, m_2(v_2)) \cdot (\mathbf{q}_1, m_1(v_1)) \cdot h_2$ whenever $\mathbf{q}_1 \neq \mathbf{q}_2$. Moreover, we consider concatenation (\cdot) as associative, and the empty queue ϵ as the identity element for concatenation.

A session, ranged over by $\mathcal{M}, \mathcal{M}'$, consists of processes and their respective incoming queue, indexed by their roles. A single entry for a role \mathbf{p} is denoted $\mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h$, where P is the process for \mathbf{p} and h is the incoming queue. Entries are composed together in parallel as $\mathcal{M} \mid \mathcal{M}'$, where the roles in \mathcal{M} and \mathcal{M}' are disjoint. We consider parallel composition as commutative and associative, with $\mathbf{p} \triangleleft \mathbf{0} \mid \mathbf{p} \triangleleft \epsilon$ as a neutral element of the operator. We write $\prod_{i \in I} (\mathbf{p}_i \triangleleft P_i \mid \mathbf{p}_i \triangleleft h_i)$ for the parallel composition of multiple entries in a set.

Operational Semantics. Operational Semantics of our session calculus is given in Def. 1, using a standard *structural congruence* \equiv defined in [16]. Our semantics parameterises on a (possibly empty) set of *reliable roles* \mathcal{R} , which are assumed to *never crash*.

² In [16], the queues are outgoing instead of incoming. We use incoming queues to model our crashing semantics more easily.

$[R-\downarrow]$	$\mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathcal{M} \rightarrow_{\mathcal{R}} \mathbf{p} \triangleleft \downarrow \mid \mathbf{p} \triangleleft \emptyset \mid \mathcal{M}$	$(P \neq \mathbf{0}, \mathbf{p} \notin \mathcal{R})$
$[R-SEND]$	$\mathbf{p} \triangleleft \mathbf{q}!m(e).P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft h_{\mathbf{q}} \mid \mathcal{M}$ $\rightarrow \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft h_{\mathbf{q}} \cdot (\mathbf{p}, m(v)) \mid \mathcal{M}$	$(e \downarrow v, h_{\mathbf{q}} \neq \emptyset)$
$[R-SEND-\downarrow]$	$\mathbf{p} \triangleleft \mathbf{q}!m(e).P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M}$	
$[R-RCV]$	$\mathbf{p} \triangleleft \sum_{i \in I} \mathbf{q}^?m_i(x_i).P_i \mid \mathbf{p} \triangleleft (\mathbf{q}, m_k(v)) \cdot h_{\mathbf{p}} \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft P_k\{v/x_k\} \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathcal{M}$	$(k \in I)$
$[R-RCV-\emptyset]$	$\mathbf{p} \triangleleft \sum_{i \in I} \mathbf{q}^?m_i(x_i).P_i \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M}$ $\rightarrow \mathbf{p} \triangleleft P_k \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathbf{q} \triangleleft \downarrow \mid \mathbf{q} \triangleleft \emptyset \mid \mathcal{M}$	$(k \in I, m_k = \text{crash}, \nexists m, v : (\mathbf{q}, m(v)) \in h_{\mathbf{p}})$
$[R-COND-T]$	$\mathbf{p} \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathbf{p} \triangleleft h \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h \mid \mathcal{M}$	$(e \downarrow \text{true})$
$[R-COND-F]$	$\mathbf{p} \triangleleft \text{if } e \text{ then } P \text{ else } Q \mid \mathbf{p} \triangleleft h \mid \mathcal{M} \rightarrow \mathbf{p} \triangleleft Q \mid \mathbf{p} \triangleleft h \mid \mathcal{M}$	$(e \downarrow \text{false})$
$[R-STRUCT]$	$\mathcal{M}_1 \equiv \mathcal{M}'_1 \text{ and } \mathcal{M}'_1 \rightarrow \mathcal{M}'_2 \text{ and } \mathcal{M}'_2 \equiv \mathcal{M}_2 \implies \mathcal{M}_1 \rightarrow \mathcal{M}_2$	

■ **Figure 5** Reduction relation on sessions with crash-stop failures.

► **Definition 1** (Session Reductions). *Session reduction $\rightarrow_{\mathcal{R}}$ is inductively defined by the rules in Fig. 5, parameterised by a fixed set \mathcal{R} of reliable roles. We write \rightarrow when \mathcal{R} is insignificant. We write $\rightarrow_{\mathcal{R}}^*$ (resp. \rightarrow^*) for the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$ (resp. \rightarrow).*

Our operational semantics retains the basic rules in [16], but also includes (highlighted) rules for crash-stop failures and crash handling, adapted from [3]. Rules $[R-SEND]$ and $[R-RCV]$ model ordinary message delivery and reception: an output process located at \mathbf{p} sending to \mathbf{q} appends a message to the incoming queue of \mathbf{q} ; and an input process located at \mathbf{p} receiving from \mathbf{q} consumes the first message from the incoming queue. Rules $[R-COND-T]$ and $[R-COND-F]$ model conditionals; and rule $[R-STRUCT]$ permits reductions up to structural congruence.

With regard to crashes and related behaviour, rule $[R-\downarrow]$ models process crashes: an active ($P \neq \mathbf{0}$) process located at an unreliable role ($\mathbf{p} \notin \mathcal{R}$) may reduce to a crashed process $\mathbf{p} \triangleleft \downarrow$, with its incoming queue becoming unavailable $\mathbf{p} \triangleleft \emptyset$. Rule $[R-SEND-\downarrow]$ models a message delivery to a crashed role (and thus an unavailable queue), and the message becomes lost and would not be added to the queue. Rule $[R-RCV-\emptyset]$ models crash detection, which activates as a “last resort”: an input process at \mathbf{p} receiving from \mathbf{q} would first attempt find a message from \mathbf{q} in the incoming queue, which engages the usual rule $[R-RCV]$; if none exists and \mathbf{q} has crashed ($\mathbf{q} \triangleleft \downarrow$), then the crash handling branch in the input process at \mathbf{p} can activate. We draw attention to the interesting fact that $[R-RCV]$ may engage even if \mathbf{q} has crashed, in cases where a message from \mathbf{q} in the incoming queue may be consumed. We now illustrate our operational semantics of sessions with an example.

► **Example 2.** Consider the session $\mathcal{M} = \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft \epsilon \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft \epsilon$, where $P = \mathbf{q}!m(\text{“abc”}). \sum \left\{ \mathbf{q}^?m'(x).\mathbf{0} \right\}$ and $Q = \sum \left\{ \mathbf{p}^?m(x).\mathbf{p}!m'(42).\mathbf{0} \right\}$. In this session, the process Q for \mathbf{q} receives a message sent from \mathbf{p} to \mathbf{q} ; the process P for \mathbf{p} sends a message from \mathbf{p} to \mathbf{q} , and then receives a message sent from \mathbf{q} to \mathbf{p} . Let each role be unreliable, i.e. $\mathcal{R} = \emptyset$, and P crash before sending. We have $\mathcal{M} \rightarrow_{\emptyset} \mathbf{p} \triangleleft \downarrow \mid \mathbf{p} \triangleleft \emptyset \mid \mathbf{q} \triangleleft Q \mid \mathbf{q} \triangleleft \epsilon \rightarrow \mathbf{p} \triangleleft \downarrow \mid \mathbf{p} \triangleleft \emptyset \mid \mathbf{q} \triangleleft \mathbf{0} \mid \mathbf{q} \triangleleft \epsilon$. We observe that when the output process P located at an unreliable role \mathbf{p} crashes (by $[R-\downarrow]$), the resulting entry for \mathbf{p} is a crashed process ($\mathbf{p} \triangleleft \downarrow$) with an unavailable queue ($\mathbf{p} \triangleleft \emptyset$). Subsequently, the input process Q located at \mathbf{q} can detect and handle the crash by $[R-RCV-\emptyset]$ via its crash handling branch.

B	$::=$	<code>int</code> <code>bool</code> <code>real</code> <code>unit</code> ...	Basic types
G	$::=$	<code>p→q†: {m_i(B_i).G_i}_{i∈I}</code>	Transmission
		<code>p†↔q:j {m_i(B_i).G_i}_{i∈I} (j ∈ I)</code>	Transmission en route
		<code>μt.G</code> <code>t</code> <code>end</code>	Recursion, Type variable, Termination
$†$	$::=$	<code>·</code> <code>⚡</code>	Crash annotation
S, T	$::=$	<code>p&{m_i(B_i).T_i}_{i∈I} <code>p⊕{m_i(B_i).T_i}_{i∈I}</code></code>	External choice, Internal choice
		<code>μt.T</code> <code>t</code> <code>end</code> <code>stop</code>	Recursion, Type variable, Termination, Crash

■ **Figure 6** Syntax of global types and local types. Runtime types are shaded.

4 Asynchronous Multiparty Session Types with Crash-Stop Semantics

In this section, we present our asynchronous multiparty session types with crash-stop semantics. We give an overview of global and local types with crashes in §4.1, including syntax, projection, subtyping, *etc.*; our key additions to the classic theory are *crash handling branches* in both global and local types, and a special local type `stop` to denote crashed processes. We give a Labelled Transition System (LTS) semantics to both global types (§4.2) and configurations (i.e. a collection of local types and point-to-point communication queues, §4.3). We discuss alternative design options of modelling crash-stop failures in §4.4. We relate the two semantics in §4.5, and show that a configuration obtained via projection is safe, deadlock-free, and live in §4.6.

4.1 Global and Local Types with Crash-Stop Failures

The top-down methodology begins with *global types* to provide an overview of the communication between a number of *roles* (`p, q, s, t, ...`), belonging to a (fixed) set \mathcal{R} . At the other end, we use *local types* to describe how a *single* role communicates with other roles from a local perspective, and they are obtained via *projection* from a global type. We give the syntax of both global and local types in Fig. 6, which are similar to syntax used in [3, 37].

Global Types. Global Types are ranged over G, G', G_i, \dots , and describe the behaviour for all roles from a bird's eye view. The syntax shown in `shade` are *runtime* syntax, which are not used for describing a system at design-time, but for describing the state of a system during execution. The labels `m` are taken from a fixed set of all labels \mathcal{M} , and basic types B (types for payloads) from a fixed set of all basic types \mathcal{B} .

We explain each construct in the syntax of global types: a transmission, denoted `p→q†: {mi(Bi).Gi}i∈I`, represents a message from role `p` to role `q` (with possible crash annotations), with labels `mi`, payload types B_i , and continuations G_i , where i is taken from an index set I . We require that the index set be non-empty ($I \neq \emptyset$), labels `mi` be pair-wise distinct, and self receptions be excluded (i.e. `p ≠ q`), as standard in session type works. Additionally, we require that the special `crash` label (explained later) not be the only label in a transmission, i.e. $\{m_i \mid i \in I\} \neq \{\text{crash}\}$. A transmission en route `p†↔q:j {mi(Bi).Gi}i∈I` is a runtime construct representing a message `mj` (index j) sent by `p`, and yet to be received by `q`. Recursive types are represented via `μt.G` and `t`, where contractive requirements apply [34, §21.8]. The type `end` describes a terminated type (omitted where unambiguous).

To model crashes and crash handling, we use crash annotations `⚡` and crash handling branches: a *crash annotation* `⚡`, a new addition in this work, marks a *crashed* role (only used in the *runtime syntax*), and we omit annotations for live roles, i.e. `p` is a live role, `p⚡` is

a crashed role, and p^\dagger represents a possibly crashed role, namely either p or p^\dagger . We use a special label `crash` for handling crashes: this continuation denotes the protocol to follow when the sender of a message is detected to have crashed by the receiver. The special label acts as a “pseudo-message”: when a sender role crashes, the receiver can select the pseudo-message to enter crash handling. We write $\text{roles}(G)$ (resp. $\text{roles}^\dagger(G)$) for the set of *active* (resp. *crashed*) roles in a global type G , *excluding* (resp. *consisting only of*) those with a crash annotation \dagger .

Local Types. Local Types are ranged over S, T, U, \dots , and describe the behaviour of a single role. An internal choice (selection) (or an external choice (branching)), denoted $p \oplus \{m_i(B_i).T_i\}_{i \in I}$ (or $p \& \{m_i(B_i).T_i\}_{i \in I}$), indicates that the *current* role is to *send* to (or *receive* from) the role p . Similarly to global types, we require pairwise-distinct, non-empty labels. Moreover, we require that the `crash` label not appear in *internal* choices, reflecting that a `crash` pseudo-message can never be sent; and that singleton `crash` labels not permitted in external choices. The type `end` indicates a *successful* termination (omitted where unambiguous), and recursive types follow a similar fashion to global types. We use a new *runtime* type `stop` to denote crashes.

Subtyping. Subtyping relation \leq on local types will be used in §4.5 to relate global and local type semantics. Our subtyping relation is mostly standard [37, Def. 2.5], except for an extra rule for `stop` and additional requirements to support crash handling branch in external choices.

Projection. Projection gives the local type of a participating role in a global type, defined as a *partial* function that takes a global type G and a role p , and returns a local type, given by Def. 3.

► **Definition 3** (Global Type Projection). *The projection of a global type G onto a role p , with respect to a set of reliable roles \mathcal{R} , written $G \upharpoonright_{\mathcal{R}} p$, is:*

$$\begin{aligned}
 (q \rightarrow r^\dagger : \{m_i(B_i).G_i\}_{i \in I}) \upharpoonright_{\mathcal{R}} p &= \begin{cases} r \oplus \{m_i(B_i).(G_i \upharpoonright_{\mathcal{R}} p)\}_{i \in \{j \in I \mid m_j \neq \text{crash}\}} & \text{if } p = q \\ q \& \{m_i(B_i).(G_i \upharpoonright_{\mathcal{R}} p)\}_{i \in I} & \text{if } p = r, \text{ and } q \notin \mathcal{R} \text{ implies } \\ & \exists k \in I : m_k = \text{crash} \\ \prod_{i \in I} G_i \upharpoonright_{\mathcal{R}} p & \text{if } p \neq q, \text{ and } p \neq r \end{cases} \\
 (q^\dagger \rightsquigarrow r : \{m_i(B_i).G_i\}_{i \in I}) \upharpoonright_{\mathcal{R}} p &= \begin{cases} G_j \upharpoonright_{\mathcal{R}} p & \text{if } p = q \\ q \& \{m_i(B_i).(G_i \upharpoonright_{\mathcal{R}} p)\}_{i \in I} & \text{if } p = r, \text{ and } q \notin \mathcal{R} \text{ implies } \\ & \exists k \in I : m_k = \text{crash} \\ \prod_{i \in I} G_i \upharpoonright_{\mathcal{R}} p & \text{if } p \neq q, \text{ and } p \neq r \end{cases} \\
 (\mu t. G) \upharpoonright_{\mathcal{R}} p &= \begin{cases} \mu t.(G \upharpoonright_{\mathcal{R}} p) & \text{if } p \in G \text{ or } \text{fv}(\mu t. G) \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} \quad \begin{matrix} t \upharpoonright_{\mathcal{R}} p = t \\ \text{end} \upharpoonright_{\mathcal{R}} p = \text{end} \end{matrix}
 \end{aligned}$$

where \prod is the merge operator for local types (full merging):

$$\begin{aligned}
 & p \& \{m_i(B_i).S'_i\}_{i \in I} \prod p \& \{m_j(B_j).T'_j\}_{j \in J} \\
 = & p \& \{m_k(B_k).(S'_k \prod T'_k)\}_{k \in I \cap J} \& p \& \{m_i(B_i).S'_i\}_{i \in I \setminus J} \& p \& \{m_j(B_j).T'_j\}_{j \in J \setminus I} \\
 & p \oplus \{m_i(B_i).S'_i\}_{i \in I} \prod p \oplus \{m_i(B_i).T'_i\}_{i \in I} = p \oplus \{m_i(B_i).(S'_i \prod T'_i)\}_{i \in I} \\
 & \mu t. S \prod \mu t. T = \mu t.(S \prod T) \quad t \prod t = t \quad \text{end} \prod \text{end} = \text{end}
 \end{aligned}$$

We parameterise our theory on a (fixed) set of *reliable* roles \mathcal{R} , i.e. roles assumed to *never crash*: if $\mathcal{R} = \emptyset$, every role is unreliable and susceptible to crash; if $\text{roles}(G) \subseteq \mathcal{R}$, every role in G is reliable, and we simulate the results from the original MPST theory without crashes. We base our definition of projection on [37], but include more (highlighted) cases to account for reliable roles, `crash` branches, and runtime global types.

When projecting a transmission from q to r , we remove the **crash** label from the internal choice at q , reflecting our model that a **crash** pseudo-message cannot be sent. Dually, we require a **crash** label to be present in the external choice at r – unless the sender role q is assumed to be reliable. Our definition of projection enforces that transmissions, whenever an unreliable role is the sender ($q \notin \mathcal{R}$), *must include* a crash handling branch ($\exists k \in I : m_k = \text{crash}$). This requirement ensures that the receiving role r can *always* handle crashes whenever it happens, so that processes are not stuck when crashes occur. We explain how these requirements help us achieve various properties by projection in §4.6. The rest of the rules are taken from the literature [37,40], without much modification.

4.2 Crash-Stop Semantics of Global Types

We now give a Labelled Transition System (LTS) semantics to global types, with crash-stop semantics. To this end, we first introduce some auxiliary definitions. We define the transition labels in Def. 4, which are also used in the LTS semantics of configurations (later in §4.3).

► **Definition 4** (Transition Labels). *Let α be a transition label of the form:*

$$\alpha ::= \begin{array}{l|l} p\&q:m(B) & (p \text{ receives } m(B) \text{ from } q) \\ p\downarrow & (p \text{ crashes}) \end{array} \quad \Bigg| \quad \begin{array}{l|l} p\oplus q:m(B) & (p \text{ sends } m(B) \text{ to } q) \\ p\odot q & (p \text{ detects the crash of } q) \end{array}$$

The subject of a transition label, written $\text{subj}(\alpha)$, is defined as:

$$\text{subj}(p\&q:m(B)) = \text{subj}(p\oplus q:m(B)) = \text{subj}(p\downarrow) = \text{subj}(p\odot q) = p.$$

The labels $p\oplus q:m(B)$ and $p\&q:m(B)$ describe sending and receiving actions respectively. The crash of p is denoted by the label $p\downarrow$, and the detection of a crash by label $p\odot q$: we model crash detection at *reception*, the label contains a *detecting* role p and a *crashed* role q .

We define an operator to *remove* a role from a global type in Def. 5: the intuition is to remove any interaction of a crashed role from the given global type. When a role has crashed, we attach a *crashed annotation*, and remove infeasible actions, e.g. when the sender and receiver of a transmission have both crashed. The removal operator is a partial function that takes a global type G and a live role r ($r \in \text{roles}(G)$) and gives a global type $G\downarrow r$.

► **Definition 5** (Role Removal). *The removal of a live role p in a global type G , written $G\downarrow p$, is defined as follows:*

$$\begin{aligned} (p \rightarrow q : \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} p\downarrow \rightsquigarrow q:j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p = r \text{ and } \exists j \in I : m_j = \text{crash} \\ p \rightarrow q\downarrow : \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } q = r \\ p \rightarrow q : \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (p \rightsquigarrow q : j \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} p\downarrow \rightsquigarrow q:j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p = r \\ G_j\downarrow r & \text{if } q = r \\ p \rightsquigarrow q:j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (p \rightarrow q\downarrow : \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} G_j\downarrow r & \text{if } p = r \text{ and } \exists j \in I : m_j = \text{crash} \\ p \rightarrow q\downarrow : \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (p\downarrow \rightsquigarrow q:j \{m_i(B_i).G_i\}_{i \in I})\downarrow r &= \begin{cases} G_j\downarrow r & \text{if } q = r \\ p\downarrow \rightsquigarrow q:j \{m_i(B_i).(G_i\downarrow r)\}_{i \in I} & \text{if } p \neq r \text{ and } q \neq r \end{cases} \\ (\mu t.G)\downarrow r &= \begin{cases} \mu t.(G\downarrow r) & \text{if } \text{fv}(\mu t.G) \neq \emptyset \text{ or } \text{roles}(G\downarrow r) \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} \\ t\downarrow r &= t & \text{end}\downarrow r &= \text{end} \end{aligned}$$

For simple cases, the removal of a role $G\downarrow r$ attaches crash annotations \downarrow on all occurrences of the removed role r throughout global type G inductively.

$$\begin{array}{c}
 \frac{\mathbf{p} \notin \mathcal{R} \quad \mathbf{p} \in \text{roles}(G) \quad G \neq \mu\mathbf{t}.G'}{\langle \mathcal{C}; G \rangle \xrightarrow{\mathbf{p}^\ddagger} \langle \mathcal{C} \cup \{\mathbf{p}\}; G \downarrow \mathbf{p} \rangle} \text{ [GR-}\ddagger\text{]} \quad \frac{\langle \mathcal{C}; G\{\mu\mathbf{t}.G/\mathbf{t}\} \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G' \rangle}{\langle \mathcal{C}; \mu\mathbf{t}.G \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G' \rangle} \text{ [GR-}\mu\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j \neq \text{crash}}{\langle \mathcal{C}; \mathbf{p} \rightarrow \mathbf{q}; \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{p} \oplus \mathbf{q}_j(B_j)} \langle \mathcal{C}; \mathbf{p} \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle} \text{ [GR-}\oplus\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j \neq \text{crash}}{\langle \mathcal{C}; \mathbf{p}^\dagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{q} \& \mathbf{m}_j(B_j)} \langle \mathcal{C}; G'_j \rangle} \text{ [GR-}\&\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j = \text{crash}}{\langle \mathcal{C}; \mathbf{p}^\ddagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{q} \circ \mathbf{p}} \langle \mathcal{C}; G'_j \rangle} \text{ [GR-}\circ\text{]} \\
 \frac{j \in I \quad \mathbf{m}_j \neq \text{crash}}{\langle \mathcal{C}; \mathbf{p} \rightarrow \mathbf{q}^\ddagger; \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\mathbf{p} \oplus \mathbf{q}_j(B_j)} \langle \mathcal{C}; G'_j \rangle} \text{ [GR-}\ddagger\mathbf{m}\text{]} \\
 \frac{\forall i \in I : \langle \mathcal{C}; G'_i \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G''_i \rangle \quad \text{subj}(\alpha) \notin \{\mathbf{p}, \mathbf{q}\}}{\langle \mathcal{C}; \mathbf{p} \rightarrow \mathbf{q}^\dagger; \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; \mathbf{p} \rightarrow \mathbf{q}^\dagger; \{ \mathbf{m}_i(B_i).G''_i \}_{i \in I} \rangle} \text{ [GR-Ctx-I]} \\
 \frac{\forall i \in I : \langle \mathcal{C}; G'_i \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; G''_i \rangle \quad \text{subj}(\alpha) \neq \mathbf{q}}{\langle \mathcal{C}; \mathbf{p}^\dagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G'_i \}_{i \in I} \rangle \xrightarrow{\alpha} \langle \mathcal{C}'; \mathbf{p}^\dagger \rightsquigarrow \mathbf{q}; j \{ \mathbf{m}_i(B_i).G''_i \}_{i \in I} \rangle} \text{ [GR-Ctx-II]}
 \end{array}$$

■ **Figure 7** Global Type Reduction Rules.

We draw attention to some interesting cases: when we remove the sender role \mathbf{p} from a transmission prefix $\mathbf{p} \rightarrow \mathbf{q}$, the result is a “pseudo-transmission” en route prefix $\mathbf{p}^\ddagger \rightsquigarrow \mathbf{q}; j$ where $\mathbf{m}_j = \text{crash}$. This enables the receiver \mathbf{q} to “receive” the special crash after the crash of \mathbf{p} , hence triggering the crash handling branch. Recall that our definition of projection requires that a crash handling branch be present whenever a crash may occur ($\mathbf{q} \notin \mathcal{R}$).

When we remove the sender role \mathbf{p} from a transmission en route prefix $\mathbf{p} \rightsquigarrow \mathbf{q}; j$, the result *retains* the index j that was selected by \mathbf{p} , instead of the index associated with crash handling. This is crucial to our crash modelling: when a role crashes, the messages that the role *has sent* to other roles are still available. We discuss alternative models later in § 4.4.

In other cases, where removing the role \mathbf{r} would render a transmission (regardless of being en route or not) meaningless, e.g. both sender and receiver have crashed, we simply remove the prefix entirely.

We now give an LTS semantics to a global type G , by defining the semantics with a tuple $\langle \mathcal{C}; G \rangle$, where \mathcal{C} is a set of *crashed* roles. The transition system is parameterised by reliability assumptions, in the form of a fixed set of reliable roles \mathcal{R} . When unambiguous, we write G as an abbreviation of $\langle \emptyset; G \rangle$. We define the reduction rules of global types in Def. 6.

► **Definition 6** (Global Type Reductions). *The global type (annotated with a set of crashed roles \mathcal{C}) transition relation $\xrightarrow{\alpha} \mathcal{R}$ is inductively defined by the rules in Fig. 7, parameterised by a fixed set \mathcal{R} of reliable roles. We write $\langle \mathcal{C}; G \rangle \rightarrow \mathcal{R} \langle \mathcal{C}'; G' \rangle$ if there exists α such that $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha} \mathcal{R} \langle \mathcal{C}'; G' \rangle$; we write $\langle \mathcal{C}; G \rangle \rightarrow \mathcal{R}$ if there exists \mathcal{C}', G' , and α such that $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha} \mathcal{R} \langle \mathcal{C}'; G' \rangle$, and $\rightarrow \mathcal{R}^*$ for the transitive and reflexive closure of $\rightarrow \mathcal{R}$.*

Rules [GR- \oplus] and [GR- $\&$] model sending and receiving messages respectively, as are standard in existing works [13]. We add an (highlighted) extra condition that the message exchanged not be a pseudo-message carrying the crash label. [GR- μ] is a standard rule handling recursion.

We introduce (highlighted) rules to account for crash and consequential behaviour. Rule [GR- \ddagger] models crashes, where a live ($\mathbf{p} \in \text{roles}(G)$), but unreliable ($\mathbf{p} \notin \mathcal{R}$) role \mathbf{p} may crash. The crashed role \mathbf{p} is added into the set of crashed roles ($\mathcal{C} \cup \{\mathbf{p}\}$), and removed from the global type, resulting in a global type $G \downarrow \mathbf{p}$. Rule [GR- \circ] is for *crash detection*, where a

live role q may detect that p has crashed at reception, and then continues with the crash handling continuation labelled `crash`. This rule only applies when the message en route is a pseudo-message, since otherwise a message rests in the queue of the receiver and can be received despite the crash of the sender (cf. $[\text{GR-}\&]$). Rule $[\text{GR-}\dagger\text{m}]$ models the orphaning of a message sent from a live role p to a crashed role q . Similar to the requirement in $[\text{GR-}\oplus]$, we add the side condition that the message sent is not a pseudo-message.

Finally, rules $[\text{GR-CTX-I}]$ and $[\text{GR-CTX-II}]$ allow non-interfering reductions of (intermediate) global types under prefix, provided that all of the continuations can be reduced by that label.

► **Remark 7 (Necessity of \mathcal{C} in Semantics).** While we can obtain the set of crashed roles in any global type G via $\text{roles}^\dagger(G)$, we need a separate \mathcal{C} for bookkeeping purposes. To illustrate, let $G = p \rightarrow q: \{m.\text{end}, \text{crash.end}\}$; we can have the following reductions:

$$\langle \emptyset; G \rangle \xrightarrow{q^\dagger} \langle \{q\}; p \rightarrow q^\dagger: \{m.\text{end}, \text{crash.end}\} \rangle \xrightarrow{p \oplus q: m} \langle \{q\}; \text{end} \rangle$$

While we can deduce q is a crashed role in the interim global type, the same information cannot be recovered from the final global type `end`.

4.3 Crash-Stop Semantics of Configurations

After giving semantics to global types, we now give an LTS semantics to *configurations*, i.e. a collection of local types and communication queues across roles. We first give a definition of configurations in Def. 8, followed by their reduction rules in Def. 9.

► **Definition 8 (Configurations).** A configuration is a tuple $\Gamma; \Delta$, where Γ is a typing context, denoting a partial mapping from roles to local types, defined as: $\Gamma ::= \emptyset \mid \Gamma, p \triangleright T$. We write $\Gamma[p \mapsto T]$ for updates: $\Gamma[p \mapsto T](p) = T$ and $\Gamma[p \mapsto T](q) = \Gamma(q)$ (where $p \neq q$).

A queue, denoted τ , is either a (possibly empty) sequence of messages $M_1 \cdot M_2 \cdots M_n$, or unavailable \circ . We write ϵ for an empty queue, and $M \cdot \tau'$ for a non-empty queue with message M at the beginning. A queue message M is of form $m(B)$, denoting a message with label m and payload B . We sometimes omit B when the payload is not of specific interest.

We write Δ to denote a queue environment, a collection of peer-to-peer queues. A queue from p to q at Δ is denoted $\Delta(p, q)$. We define updates $\Delta[p, q \mapsto \tau]$ similarly. We write Δ_\emptyset for an empty queue environment, where $\Delta_\emptyset(p, q) = \epsilon$ for any p and q in the domain.

We write $\tau' \cdot M$ to append a message M at the end of a queue τ' : the message is appended to the sequence when τ' is available, or discarded when τ' is unavailable (i.e. $\circ \cdot M = \circ$). Additionally, we write $\Delta[\cdot, q \mapsto \circ]$ for making all the queues to q unavailable: i.e. $\Delta[p_1, q \mapsto \circ][p_2, q \mapsto \circ] \cdots [p_n, q \mapsto \circ]$.

We give an LTS semantics of configurations in Def. 9. Similar to that of global types, we model the semantics of configurations in an asynchronous (a.k.a. message passing) fashion, using a queue environment to represent the communication queues among all roles.

► **Definition 9 (Configuration Semantics).** The configuration transition relation $\xrightarrow{\alpha}$ is defined in Fig. 8. We write $\Gamma; \Delta \xrightarrow{\alpha}$ iff $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ for some Γ' and Δ' . We define two reductions \rightarrow and $\rightarrow_{\mathcal{R}}$ (where \mathcal{R} is a fixed set of reliable roles) as follows:

- We write $\Gamma; \Delta \rightarrow \Gamma'; \Delta'$ for $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ with $\alpha \in \{p \& q: m(B), p \oplus q: m(B), p \circ q\}$. We write $\Gamma; \Delta \rightarrow$ iff $\Gamma; \Delta \rightarrow \Gamma'; \Delta'$ for some $\Gamma'; \Delta'$, and $\Gamma; \Delta \not\rightarrow$ for its negation, and \rightarrow^* for the reflexive and transitive closure of \rightarrow ;
- We write $\Gamma; \Delta \rightarrow_{\mathcal{R}} \Gamma'; \Delta'$ for $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$ with $\alpha \notin \{r^\dagger \mid r \in \mathcal{R}\}$. We write $\Gamma; \Delta \rightarrow_{\mathcal{R}}$ iff $\Gamma; \Delta \rightarrow_{\mathcal{R}} \Gamma'; \Delta'$ for some $\Gamma'; \Delta'$, and $\Gamma; \Delta \not\rightarrow_{\mathcal{R}}$ for its negation. We define $\rightarrow_{\mathcal{R}}^*$ as the reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$.

$$\begin{array}{c}
\frac{\Gamma(\mathbf{p}) = \mathbf{q} \oplus \{\mathbf{m}_i(B_i).T_i\}_{i \in I} \quad k \in I}{\Gamma; \Delta \xrightarrow{\mathbf{p} \oplus \mathbf{q}; \mathbf{m}_k(B_k)} \Gamma[\mathbf{p} \mapsto T_k]; \Delta[\mathbf{p}, \mathbf{q} \mapsto \Delta(\mathbf{p}, \mathbf{q}) \cdot \mathbf{m}_k(B_k)]} \text{[}\Gamma\text{-}\oplus\text{]} \\
\frac{\Gamma(\mathbf{p}) = \mathbf{q} \& \{\mathbf{m}_i(B_i).T_i\}_{i \in I} \quad k \in I \quad \mathbf{m}' = \mathbf{m}_k \quad B' = B_k \quad \Delta(\mathbf{q}, \mathbf{p}) = \mathbf{m}'(B') \cdot \tau' \neq \emptyset}{\Gamma; \Delta \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}_k(B_k)} \Gamma[\mathbf{p} \mapsto T_k]; \Delta[\mathbf{q}, \mathbf{p} \mapsto \tau']} \text{[}\Gamma\text{-}\&\text{]} \\
\frac{\Gamma(\mathbf{p}) = \mu t. T \quad \Gamma[\mathbf{p} \mapsto T\{\mu t. T/t\}]; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'}{\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'} \text{[}\Gamma\text{-}\mu\text{]} \quad \frac{\Gamma(\mathbf{p}) \neq \text{end} \quad \Gamma(\mathbf{p}) \neq \text{stop}}{\Gamma; \Delta \xrightarrow{\mathbf{p} \ddagger} \Gamma[\mathbf{p} \mapsto \text{stop}]; \Delta[\cdot, \mathbf{p} \mapsto \emptyset]} \text{[}\Gamma\text{-}\ddagger\text{]} \\
\frac{\Gamma(\mathbf{q}) = \mathbf{p} \& \{\mathbf{m}_i(B_i).T_i\}_{i \in I} \quad \Gamma(\mathbf{p}) = \text{stop} \quad k \in I \quad \mathbf{m}_k = \text{crash} \quad \Delta(\mathbf{p}, \mathbf{q}) = \epsilon}{\Gamma; \Delta \xrightarrow{\mathbf{q} \odot \mathbf{p}} \Gamma[\mathbf{q} \mapsto T_k]; \Delta} \text{[}\Gamma\text{-}\odot\text{]}
\end{array}$$

■ **Figure 8** Configuration Semantics.

We first explain the standard rules: rule $[\Gamma\text{-}\oplus]$ (resp. $[\Gamma\text{-}\&]$) says that a role can perform an output (resp. input) transition by appending (resp. consuming) a message at the corresponding queue. Recall that whenever a queue is unavailable, the resulting queue remains unavailable after appending ($\emptyset \cdot M = \emptyset$). Therefore, the rule $[\Gamma\text{-}\oplus]$ covers delivery to both crashed and live roles, whereas two separate rules are used in modelling global type semantics ($[\text{GR}\text{-}\oplus]$ and $[\text{GR}\text{-}\ddagger\text{m}]$). We also include a standard rule $[\Gamma\text{-}\mu]$ for recursive types.

The key innovations are the (highlighted) rules modelling crashes and crash detection: by rule $[\Gamma\text{-}\ddagger]$, a role \mathbf{p} may crash and become **stop** at any time (unless it is already **ended** or **stopped**). All of \mathbf{p} 's receiving queues become unavailable \emptyset , so that future messages to \mathbf{p} would be discarded. Rule $[\Gamma\text{-}\odot]$ models crash detection and handling: if \mathbf{p} is crashed and stopped, another role \mathbf{q} attempting to receive from \mathbf{p} can then take its **crash** handling branch. However, this rule only applies when the corresponding queue is empty: it is still possible to receive messages sent before crashing via $[\Gamma\text{-}\&]$.

4.4 Alternative Modellings for Crash-Stop Failures

Before we dive into the relation between two semantics, let us have a short digression to discuss our modelling choices and alternatives. In this work, we mostly follow the assumptions laid out in [3], where a crash is detected at reception. However, they opt to use a synchronous (rendez-vous) semantics, whereas we give an asynchronous (message passing) semantics, which entails interesting scenarios that would not arise in a synchronous semantics.

Specifically, consider the case where a role \mathbf{p} sends a message to \mathbf{q} , and then \mathbf{p} crashes after sending, but before \mathbf{q} receives the message. The situation does not arise under a synchronous semantics, since sending and receiving actions are combined into a single transmission action.

Intuitively, there are two possibilities to handle this scenario. The questions are whether the message sent immediately before crashing is deliverable to \mathbf{q} , and consequentially, at what time does \mathbf{q} detect the crash of \mathbf{p} .

In our semantics (Figs. 7 and 8), we opt to answer the first question in positive: we argue that this model is more consistent with our “passive” crash detection design. For example, if a role \mathbf{p} never receives from another role \mathbf{q} , then \mathbf{p} does not need to react in the event of \mathbf{q} 's crash. Following a similar line of reasoning, if the message sent by \mathbf{p} arrives in the receiving queue of \mathbf{q} , then \mathbf{q} should be able to receive the message, without triggering a crash detection (although it may be triggered later). As a consequence, we require in $[\Gamma\text{-}\odot]$ that the queue $\Delta(\mathbf{p}, \mathbf{q})$ be empty, to reflect the idea that crash detection should be a “last resort”.

For an alternative model, we can opt to detect the crash after crash has occurred. This is possibly better modelled with using outgoing queues (cf. [12]), instead of incoming queues in the semantics presented. Practically, this may be the scenario that a TCP connection is closed (or reset) when a peer has crashed, and the content in the queue is lost. It is worth noting that this kind of alternative model will not affect our main theoretical results: the operational correspondence between global and local type semantics, and furthermore, global type properties guaranteed by projection.

4.5 Relating Global Type and Configuration Semantics

We have given LTS semantics for both global types (Def. 6) and configurations (Def. 9), we now relate these two semantics with the help of the projection operator \uparrow (Def. 3).

We associate configurations $\Gamma; \Delta$ with global types G (as annotated with a set of crashed roles \mathcal{C}) by projection, written $\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$. Naturally, there are two components of the association: (1) the local types in Γ need to correspond to the projections of the global type G and the set of crashed roles \mathcal{C} ; and (2) the queues in Δ corresponds to the transmissions en route in the global type G and also the set of crashed roles \mathcal{C} .

► **Definition 10** (Association of Global Types and Configurations). *A configuration $\Gamma; \Delta$ is associated with a (well-annotated w.r.t. \mathcal{R}) global type $\langle \mathcal{C}; G \rangle$, written $\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$, iff*

1. Γ can be split into disjoint (possibly empty) sub-contexts $\Gamma = \Gamma_G, \Gamma_{\mathcal{C}}, \Gamma_{\text{end}}$ where:
 - (A1) Γ_G contains projections of G : $\text{dom}(\Gamma_G) = \text{roles}(G)$, and $\forall \mathbf{p} \in \text{dom}(\Gamma_G) : \Gamma(\mathbf{p}) \leq G \uparrow_{\mathcal{R}} \mathbf{p}$;
 - (A2) $\Gamma_{\mathcal{C}}$ contains crashed roles: $\text{dom}(\Gamma_{\mathcal{C}}) = \mathcal{C}$, and $\forall \mathbf{p} \in \text{dom}(\Gamma_{\mathcal{C}}) : \Gamma(\mathbf{p}) = \text{stop}$;
 - (A3) Γ_{end} contains only end endpoints: $\forall \mathbf{p} \in \Gamma_{\text{end}} : \Gamma(\mathbf{p}) = \text{end}$.
2. (A4) Δ is associated with global type $\langle \mathcal{C}; G \rangle$, given as follows:
 - i. Receiving queues for a role is unavailable if and only if it has crashed: $\forall \mathbf{q} : \mathbf{q} \in \mathcal{C} \iff \Delta(\cdot, \mathbf{q}) = \emptyset$;
 - ii. If $G = \text{end}$ or $G = \mu \mathbf{t}.G'$, then queues between all roles are empty (except receiving queue for crashed roles): $\forall \mathbf{p}, \mathbf{q} : \mathbf{q} \notin \mathcal{C} \implies \Delta(\mathbf{p}, \mathbf{q}) = \epsilon$;
 - iii. If $G = \mathbf{p} \rightarrow \mathbf{q}^{\dagger} : \{m_i(B_i).G'_i\}_{i \in I}$, or $G = \mathbf{p}^{\dagger} \rightsquigarrow \mathbf{q} : j \{m_i(B_i).G'_i\}_{i \in I}$ with $m_j = \text{crash}$ (i.e. a pseudo-message is en route), then
 - (i) if \mathbf{q} is live, then the queue from \mathbf{p} to \mathbf{q} is empty: $\mathbf{q}^{\dagger} \neq \mathbf{q}^{\ddagger} \implies \Delta(\mathbf{p}, \mathbf{q}) = \epsilon$, and
 - (ii) $\forall i \in I : \Delta$ is associated with $\langle \mathcal{C}; G'_i \rangle$;
 - iv. If $G = \mathbf{p}^{\dagger} \rightsquigarrow \mathbf{q} : j \{m_i(B_i).G'_i\}_{i \in I}$ with $m_j \neq \text{crash}$, then
 - (i) the queue from \mathbf{p} to \mathbf{q} begins with the message $m_j(B_j) : \Delta(\mathbf{p}, \mathbf{q}) = m_j(B_j) \cdot \tau$;
 - (ii) $\forall i \in I : \text{removing the message from the head of the queue, } \Delta[\mathbf{p}, \mathbf{q} \mapsto \tau]$ is associated with $\langle \mathcal{C}; G'_i \rangle$.

We write $\Gamma \sqsubseteq_{\mathcal{R}} G$ as an abbreviation of $\Gamma; \Delta_{\emptyset} \sqsubseteq_{\mathcal{R}} \langle \emptyset; G \rangle$. We sometimes say Γ (resp. Δ) is associated with $\langle \mathcal{C}; G \rangle$ for stating Item 1 (resp. Item 2) is satisfied.

We demonstrate the relation between the two semantics via association, by showing two main theorems: all possible reductions of a configuration have a corresponding action in reductions of the associated global type (Thm. 11); and the reducibility of a global type is the same as its associated configuration (Thm. 12).

► **Theorem 11** (Completeness of Association). *Given associated global type G and configuration $\Gamma; \Delta : \Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$. If $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$, where $\alpha \neq \mathbf{p}^{\ddagger}$ for all $\mathbf{p} \in \mathcal{R}$, then there exists $\langle \mathcal{C}'; G' \rangle$ such that $\Gamma'; \Delta' \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$ and $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha}_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$.*

► **Theorem 12** (Soundness of Association). *Given associated global type G and configuration $\Gamma; \Delta: \Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$. If $\langle \mathcal{C}; G \rangle \rightarrow_{\mathcal{R}}$, then there exists $\Gamma'; \Delta'$, α and $\langle \mathcal{C}'; G' \rangle$, such that $\langle \mathcal{C}; G \rangle \xrightarrow{\alpha}_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$, $\Gamma'; \Delta' \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}'; G' \rangle$, and $\Gamma; \Delta \xrightarrow{\alpha} \Gamma'; \Delta'$.*

By Thms. 11 and 12, we obtain, as a corollary, that a global type G is in operational correspondence with the typing context $\Gamma = \{\mathbf{p} \triangleright G \upharpoonright_{\mathcal{R}} \mathbf{p}\}_{\mathbf{p} \in \text{roles}(G)}$, which contains the projections of all roles in G .

4.6 Properties Guaranteed by Projection

A key benefit of our top-down approach of multiparty protocol design is that desirable properties are guaranteed by the methodology. As a consequence, processes following the local types obtained from projections are correct *by construction*. In this subsection, we focus on three properties: *communication safety*, *deadlock-freedom*, and *liveness*, and show that the three properties are guaranteed from configurations associated with global types.

Communication Safety. We begin by defining communication safety for configurations (Def. 13). We focus on two safety requirements:

- (i) each role must be able to handle any message that may end up in their receiving queue (so that there are no label mismatches); and
- (ii) each receiver must be able to handle the potential crash of the sender, unless the sender is reliable.

► **Definition 13** (Configuration Safety). *Given a fixed set of reliable roles \mathcal{R} , we say that φ is an \mathcal{R} -safety property of configurations iff, whenever $\varphi(\Gamma; \Delta)$, we have:*

[S- $\oplus \&$] $\Gamma(\mathbf{q}) = \mathbf{p} \& \{m_i(B_i).S'_i\}_{i \in I}$ and $\Delta(\mathbf{p}, \mathbf{q}) \neq \emptyset$ and $\Delta(\mathbf{p}, \mathbf{q}) \neq \epsilon$ implies $\Gamma; \Delta \xrightarrow{\mathbf{q} \& \mathbf{p}: m'(B')}$;

[S- $\& \&$] $\Gamma(\mathbf{p}) = \text{stop}$ and $\Gamma(\mathbf{q}) = \mathbf{p} \& \{m_i(S_i).S'_i\}_{i \in I}$ and $\Delta(\mathbf{p}, \mathbf{q}) = \epsilon$ implies $\Gamma; \Delta \xrightarrow{\mathbf{q} \oplus \mathbf{p}}$;

[S- μ] $\Gamma(\mathbf{p}) = \mu t.S$ implies $\varphi(\Gamma[\mathbf{p} \mapsto S\{\mu t.S/t\}]; \Delta)$;

[S- \rightarrow_i] $\Gamma; \Delta \rightarrow_{\mathcal{R}} \Gamma'; \Delta'$ implies $\varphi(\Gamma'; \Delta')$.

We say $\Gamma; \Delta$ is \mathcal{R} -safe, if $\varphi(\Gamma; \Delta)$ holds for some \mathcal{R} -safety property φ .

We use a coinductive view of the safety property [35], where the predicate of \mathcal{R} -safe configurations is the largest \mathcal{R} -safety property, by taking the union of all safety properties φ . For a configuration $\Gamma; \Delta$ to be \mathcal{R} -safe, it has to satisfy all clauses defined in Def. 13.

By clause [S- $\oplus \&$], whenever a role \mathbf{q} receives from another role \mathbf{p} , and a message is present in the queue, the receiving action must be possible for some label m' . Clause [S- $\& \&$] states that if a role \mathbf{q} receives from a crashed role \mathbf{p} , and there is nothing in the queue, then \mathbf{q} must have a *crash* branch, and a crash detection action can be fired. (Note that [S- $\oplus \&$] applies when the queue is non-empty, despite the crash of sender \mathbf{p} .) Finally, clause [S- μ] extends the previous clauses by unfolding any recursive entries; and clause [S- \rightarrow_i] states that any configuration $\Gamma'; \Delta'$ which $\Gamma; \Delta$ transitions to must also be \mathcal{R} -safe. By using transition $\rightarrow_{\mathcal{R}}$, we ignore crash transitions $\mathbf{p} \downarrow$ for any reliable role $\mathbf{p} \in \mathcal{R}$.

► **Example 14.** Recall the local types $T_{\mathbf{C}}$, $T_{\mathbf{L}}$, and $T_{\mathbf{I}}$ of the Simpler Logging example in § 2. The configuration $\Gamma; \Delta$, where $\Gamma = \mathbf{C} \triangleright T_{\mathbf{C}}, \mathbf{L} \triangleright T_{\mathbf{L}}, \mathbf{I} \triangleright T_{\mathbf{I}}$ and $\Delta = \Delta_{\emptyset}$, is $\{\mathbf{L}, \mathbf{I}\}$ -safe. This can be verified by checking its reductions. For example, in the case where \mathbf{C} crashes immediately, we have: $\Gamma; \Delta \xrightarrow{\mathbf{C} \downarrow} \Gamma[\mathbf{C} \mapsto \text{stop}]; \Delta[\cdot, \mathbf{C} \mapsto \emptyset] \xrightarrow{*} \Gamma[\mathbf{C} \mapsto \text{stop}][\mathbf{L} \mapsto \text{end}][\mathbf{I} \mapsto \text{end}]; \Delta[\cdot, \mathbf{C} \mapsto \emptyset]$ and each reductum satisfies all clauses of Def. 13.

Deadlock-Freedom. The property of deadlock-freedom, sometimes also known as progress, describes whether a configuration can keep reducing unless it is a terminal configuration. We give its formal definition in Def. 15.

► **Definition 15** (Configuration Deadlock-Freedom). *Given a set of reliable roles \mathcal{R} , we say that a configuration $\Gamma; \Delta$ is \mathcal{R} -deadlock-free iff:*

1. $\Gamma; \Delta$ is \mathcal{R} -safe; and,
2. If $\Gamma; \Delta$ can reduce to a configuration $\Gamma'; \Delta'$ without further reductions: $\Gamma; \Delta \rightarrow_{\mathcal{R}}^* \Gamma'; \Delta' \not\rightarrow_{\mathcal{R}}$, then:
 - a. Γ' can be split into two disjoint contexts, one with only **end** entries, and one with only **stop** entries: $\Gamma' = \Gamma'_{\text{end}}, \Gamma'_{\neq}$, where $\text{dom}(\Gamma'_{\text{end}}) = \{\mathbf{p} \mid \Gamma'(\mathbf{p}) = \text{end}\}$ and $\text{dom}(\Gamma'_{\neq}) = \{\mathbf{p} \mid \Gamma'(\mathbf{p}) = \text{stop}\}$; and,
 - b. Δ' is empty for all pairs of roles, except for the receiving queues of crashed roles, which are unavailable: $\forall \mathbf{p}, \mathbf{q} : \Delta'(\cdot, \mathbf{q}) = \emptyset$ if $\Gamma'(\mathbf{q}) = \text{stop}$, and $\Delta'(\mathbf{p}, \mathbf{q}) = \epsilon$, otherwise.

It is worth noting that a (safe) configuration that reduces infinitely satisfies deadlock-freedom, as Item 2 in the premise does not hold. Otherwise, whenever a terminal configuration is reached, it must satisfy Item 2a that all local types in the typing context be terminated (either successfully **end**, or crashed **stop**), and Item 2b that all queues be empty (unless unavailable due to crash). As a consequence, a deadlock-free configuration $\Gamma; \Delta$ either does not stop reducing, or terminates in a stable configuration.

Liveness. The property of liveness describes that every pending output/external choice is eventually triggered by means of a message transmission or crash detection. Our liveness property is based on *fairness*, which guarantees that every enabled message transmission, including crash detection, is performed successfully. We give the definitions of non-crashing, fair, and live paths of configurations respectively in Def. 16, and use these paths to formalise the liveness for configurations in Def. 17.

► **Definition 16** (Non-crashing, Fair, Live Paths). *A non-crashing path is a possibly infinite sequence of configurations $(\Gamma_n; \Delta_n)_{n \in N}$, where $N = \{0, 1, 2, \dots\}$ is a set of consecutive natural numbers, and $\forall n \in N, \Gamma_n; \Delta_n \rightarrow \Gamma_{n+1}; \Delta_{n+1}$. We say that a non-crashing path $(\Gamma_n; \Delta_n)_{n \in N}$ is fair iff, $\forall n \in N$:*

- (F1) $\Gamma_n; \Delta_n \xrightarrow{\mathbf{p} \oplus \mathbf{q}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$ implies $\exists k, \mathbf{m}', B'$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \oplus \mathbf{q}; \mathbf{m}'(B')} \Gamma_{k+1}; \Delta_{k+1}$;
- (F2) $\Gamma_n; \Delta_n \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$ implies $\exists k$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$;
- (F3) $\Gamma_n; \Delta_n \xrightarrow{\mathbf{p} \odot \mathbf{q}} \Gamma_{k+1}; \Delta_{k+1}$ implies $\exists k$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \odot \mathbf{q}} \Gamma_{k+1}; \Delta_{k+1}$.

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

- (L1) $\Delta_n(\mathbf{p}, \mathbf{q}) = \mathbf{m}(B) \cdot \tau \neq \emptyset$ and $\mathbf{m} \neq \text{crash}$ implies $\exists k$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{q} \& \mathbf{p}; \mathbf{m}(B)} \Gamma_{k+1}; \Delta_{k+1}$;
- (L2) $\Gamma_n(\mathbf{p}) = \mathbf{q} \& \{ \mathbf{m}_i(B_i) \cdot T_i \}_{i \in I}$ implies $\exists k, \mathbf{m}', B'$ such that $n \leq k \in N$ and $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \& \mathbf{q}; \mathbf{m}'(B')} \Gamma_{k+1}; \Delta_{k+1}$ or $\Gamma_k; \Delta_k \xrightarrow{\mathbf{p} \odot \mathbf{q}} \Gamma_{k+1}; \Delta_{k+1}$.

A non-crash path is a (possibly infinite) sequence of reductions of a configuration without crashes. A non-crash path is fair if along the path, every internal choice eventually sends a message (F1), every external choice eventually receives a message (F2), and every crash detection is eventually performed (F3). A non-crashing path is live if along the path, every non-crash message in the queue is eventually consumed (L1), and every hanging external choice eventually consumes a message or performs a crash detection (L2).

$$\begin{array}{c}
 \frac{}{\vdash \epsilon : \epsilon} \text{[T-}\epsilon\text{]} \quad \frac{}{\vdash \emptyset : \emptyset} \text{[T-}\emptyset\text{]} \quad \frac{\vdash h_1 : \delta_1 \quad \vdash h_2 : \delta_2}{\vdash h_1 \cdot h_2 : \delta_1 \cdot \delta_2} \text{[T-}\cdot\text{]} \\
 \frac{\vdash v : B \quad \delta(\mathbf{q}) = \mathbf{m}(B) \quad \forall \mathbf{r} \neq \mathbf{q} : \delta(\mathbf{r}) = \epsilon}{\vdash (\mathbf{q}, \mathbf{m}(v)) : \delta} \text{[T-MSG]} \\
 \\
 \frac{\frac{}{\Theta \vdash \zeta : \text{stop}} \text{[T-}\zeta\text{]} \quad \frac{}{\Theta \vdash \mathbf{0} : \text{end}} \text{[T-}\mathbf{0}\text{]} \quad \frac{\Theta \vdash e : B \quad \Theta \vdash P : T}{\Theta \vdash \mathbf{q}\mathbf{m}(e).P : \mathbf{q}\oplus\mathbf{m}(B).T} \text{[T-OUT]}}{\frac{\forall i \in I \quad \Theta, x_i : B_i \vdash P_i : T_i}{\Theta \vdash \sum_{i \in I} \mathbf{q}^{\mathbf{m}_i(x_i)}.P_i : \mathbf{q}\&\{\mathbf{m}_i(B_i).T_i\}_{i \in I}} \text{[T-EXT]} \quad \frac{\Theta \vdash e : \text{bool} \quad \Theta \vdash P_i : T \quad (i = 1, 2)}{\Theta \vdash \text{if } e \text{ then } P_1 \text{ else } P_2 : T} \text{[T-COND]}}{\frac{\frac{\Theta, X : T \vdash P : T}{\Theta \vdash \mu X.P : T} \text{[T-REC]} \quad \frac{}{\Theta, X : T \vdash X : T} \text{[T-VAR]} \quad \frac{\Theta \vdash P : T \quad T \leq T'}{\Theta \vdash P : T'} \text{[T-SUB]}}{\frac{\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle \quad \forall i \in I \quad \vdash P_i : \Gamma(\mathbf{p}_i) \quad \vdash h_i : \Delta(-, \mathbf{p}_i) \quad \text{dom}(\Gamma) \subseteq \{\mathbf{p}_i \mid i \in I\}}{\langle \mathcal{C}; G \rangle \vdash \prod_{i \in I} (\mathbf{p}_i \triangleleft P_i \mid \mathbf{p}_i \triangleleft h_i)} \text{[T-SESS]}}}
 \end{array}$$

■ **Figure 9** Typing rules for queues, processes, and sessions.

► **Definition 17** (Configuration Liveness). *Given a set of reliable roles \mathcal{R} , we say that a configuration $\Gamma; \Delta$ is \mathcal{R} -live iff:*

1. $\Gamma; \Delta$ is \mathcal{R} -safe; and,
2. $\Gamma; \Delta \rightarrow_{\mathcal{R}}^* \Gamma'; \Delta'$ implies all non-crashing paths starting with $\Gamma'; \Delta'$ that are fair are also live.

A configuration $\Gamma; \Delta$ is \mathcal{R} -live when it is \mathcal{R} -safe and any reductum of $\Gamma; \Delta$ (via transition $\rightarrow_{\mathcal{R}}^*$) consistently leads to a live path if it is fair.

Properties by Projection. We conclude by showing the guarantee of safety, deadlock-freedom, and liveness in configurations associated with global types in Lem. 18. Furthermore, as a corollary, Thm. 19 demonstrates that a typing context projected from a global type (without runtime constructs) is inherently safe, deadlock-free, and live by construction.

► **Lemma 18.** *If $\Gamma; \Delta \sqsubseteq_{\mathcal{R}} \langle \mathcal{C}; G \rangle$, then $\Gamma; \Delta$ is \mathcal{R} -safe, \mathcal{R} -deadlock-free, and \mathcal{R} -live.*

► **Theorem 19** (Safety, Deadlock-Freedom, and Liveness by Projection). *Let G be a global type without runtime constructs, and \mathcal{R} be a set of reliable roles. If Γ is a typing context associated with the global type $G: \Gamma \sqsubseteq_{\mathcal{R}} G$, then $\Gamma; \Delta_{\emptyset}$ is \mathcal{R} -safe, \mathcal{R} -deadlock-free, and \mathcal{R} -live.*

5 Typing System with Crash-Stop Semantics

In this section, we present a type system for our asynchronous multiparty session calculus. Our typing system is extended from the one in [16] with crash-stop failures. We introduce the typing rules in § 5.1, and show various properties of typed sessions: subject reduction, session fidelity, deadlock-freedom, and liveness in § 5.2.

5.1 Typing Rules

Our type system uses three kinds of typing judgements: (1) for processes; (2) for queues; and (3) for sessions, and is defined inductively by the typing rules in Fig. 9. Typing judgments for processes are of form $\Theta \vdash P : T$, where Θ is a typing context for variables, defined as $\Theta ::= \emptyset \mid \Theta, x : B \mid \Theta, X : T$.

With regard to queues, we use judgments of the form $\vdash h : \delta$, where we use δ to denote a partially applied queue lookup function. We write $\delta = \Delta(-, \mathbf{p})$ to describe the incoming queue for a role \mathbf{p} , as a partially applied function $\delta = \Delta(-, \mathbf{p})$ such that $\delta(\mathbf{q}) = \Delta(\mathbf{q}, \mathbf{p})$. We write $\delta_1 \cdot \delta_2$ to denote the point-wise application of concatenation. For empty queues (ϵ), unavailable queues (\emptyset), and queue concatenations (\cdot), we simply lift the process-level queue constructs to type-level counterparts. For a singleton message $(\mathbf{q}, \mathbf{m}(v))$, the appropriate partial queue δ would be a singleton of $\mathbf{m}(B)$ (where B is the type of v) for \mathbf{q} , and an empty queue (ϵ) for any other role.

Finally, we use judgments of the form $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$ for sessions. We use a global type-guided judgment, effectively asserting that all participants in the session respect the prescribed global type, as is the case in [15]. As **highlighted**, the global type with crashed roles $\langle \mathbf{C}; G \rangle$ must have some associated configuration $\Gamma; \Delta$, used to type the processes and the queues respectively. Moreover, all the entries in the configuration must be present in the session.

Most rules in Fig. 9 assign the corresponding session type according to the behaviour of the process. For example, (**highlighted**) rule $[\text{T-}\emptyset]$ assigns the unavailable queue type \emptyset to a unavailable queue \emptyset ; rules $[\text{T-OUT}]$ and $[\text{T-EXT}]$ assign internal and external choice types to input and output processes; (**highlighted**) rule $[\text{T-}\dagger]$ (resp. $[\text{T-}\mathbf{0}]$) assigns the crash termination **stop** (resp. successful termination **end**) to a crashed process \dagger (resp. inactive process $\mathbf{0}$).

► **Example 20.** Consider the process that acts as the role \mathbf{C} in our Simpler Logging example (§2 and Ex. 14): $P_{\mathbf{C}} = \mathbf{I!read.I?report}(x).\mathbf{0}$, and a message queue $h_{\mathbf{C}} = \epsilon$. Process $P_{\mathbf{C}}$ has the type $T_{\mathbf{C}}$, and queue $h_{\mathbf{C}}$ has the type ϵ , which can be verified in the standard way. If we follow a crash reduction, e.g. by the rule $[\text{R-}\dagger]$, the session evolves as $\mathbf{C} \triangleleft P_{\mathbf{C}} \mid \mathbf{C} \triangleleft h_{\mathbf{C}} \rightarrow_{\mathcal{R}} \mathbf{C} \triangleleft \dagger \mid \mathbf{C} \triangleleft \emptyset$, where, by $[\text{T-}\dagger]$, $P_{\mathbf{C}}$ is typed by **stop**, and $h_{\mathbf{C}}$ is typed by \emptyset .

5.2 Properties of Typed Sessions

We present the main properties of typed sessions: *subject reduction* (Thm. 21), *session fidelity* (Thm. 22), *deadlock-freedom* (Thm. 24), and *liveness* (Thm. 26).

Subject reduction states that well-typedness of sessions are preserved by reduction. In other words, a session governed by a global type continues to be governed by a global type.

► **Theorem 21 (Subject Reduction).** *If $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$ and $\mathcal{M} \rightarrow_{\mathcal{R}} \mathcal{M}'$, then either $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}'$, or there exists $\langle \mathbf{C}'; G' \rangle$ such that $\langle \mathbf{C}; G \rangle \rightarrow_{\mathcal{R}} \langle \mathbf{C}'; G' \rangle$ and $\langle \mathbf{C}'; G' \rangle \vdash \mathcal{M}'$.*

Session fidelity states the opposite implication with regard to subject reduction: sessions respect the progress of the governing global type.

► **Theorem 22 (Session Fidelity).** *If $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$ and $\langle \mathbf{C}; G \rangle \rightarrow_{\mathcal{R}}$, then there exists \mathcal{M}' and $\langle \mathbf{C}'; G' \rangle$ such that $\langle \mathbf{C}; G \rangle \rightarrow_{\mathcal{R}} \langle \mathbf{C}'; G' \rangle$, $\mathcal{M} \rightarrow_{\mathcal{R}^*} \mathcal{M}'$ and $\langle \mathbf{C}'; G' \rangle \vdash \mathcal{M}'$.*

Session *deadlock-freedom* means that the “successful” termination of a session may include crashed processes and their respective unavailable incoming queues – but reliable roles (which cannot crash) can only successfully terminate by reaching inactive processes with empty incoming queues. We formalise the definition of deadlock-free sessions in Def. 23 and show that a well-typed session is deadlock-free in Thm. 24.

► **Definition 23 (Deadlock-Free Sessions).** *A session \mathcal{M} is deadlock-free iff $\mathcal{M} \rightarrow_{\mathcal{R}^*} \mathcal{M}' \not\rightarrow_{\mathcal{R}}$ implies either $\mathcal{M}' \equiv \mathbf{p} \triangleleft \mathbf{0} \mid \mathbf{p} \triangleleft \epsilon$, or $\mathcal{M}' \equiv \mathbf{p} \triangleleft \dagger \mid \mathbf{p} \triangleleft \emptyset$.*

► **Theorem 24 (Session Deadlock-Freedom).** *If $\langle \mathbf{C}; G \rangle \vdash \mathcal{M}$, then \mathcal{M} is deadlock-free.*

```

1 global protocol SimpleLogger(role U, reliable role L)
2 { rec t0 { choice at U { write(String) from U to L;
3     continue t0; }
4     or { read from U to L;
5     report(Log) from L to U;
6     continue t0; }
7     or { crash from U to L; } } }

```

■ **Figure 10** A Simple Logger protocol in SCRIBBLE.

Finally, we show that well-typed sessions guarantee the property of *liveness*: a session is *live* when all its input processes will be performed eventually, and all its queued messages will be consumed eventually. We formalise the definition of live sessions in Def. 25 and conclude by showing that a well-typed session is live in Thm. 26.

► **Definition 25** (Live Sessions). *A session \mathcal{M} is live iff $\mathcal{M} \rightarrow_{\mathcal{R}^*} \mathcal{M}' \equiv \mathbf{p} \triangleleft P \mid \mathbf{p} \triangleleft h_{\mathbf{p}} \mid \mathcal{M}''$ implies:*

1. if $h_{\mathbf{p}} = (\mathbf{q}, \mathbf{m}(v)) \cdot h'_{\mathbf{p}}$, then $\exists P', \mathcal{M}''' : \mathcal{M}' \rightarrow_{\mathcal{R}^*} \mathbf{p} \triangleleft P' \mid \mathbf{p} \triangleleft h'_{\mathbf{p}} \mid \mathcal{M}'''$; and
2. if $P = \sum_{i \in I} \mathbf{q}^? \mathbf{m}_i(x_i).P_i$, then $\exists k \in I, w, h'_{\mathbf{p}}, \mathcal{M}''' : \mathcal{M}' \rightarrow_{\mathcal{R}^*} \mathbf{p} \triangleleft P_k \{w/x_k\} \mid \mathbf{p} \triangleleft h'_{\mathbf{p}} \mid \mathcal{M}'''$.

► **Theorem 26** (Session Liveness). *If $\langle \mathcal{C}; G \rangle \vdash \mathcal{M}$, then \mathcal{M} is live.*

6 Teatrino: Generating Scala Programs from Protocols

In this section, we present our toolchain TEATRINO that implements our extended MPST theory with crash-stop failures. TEATRINO processes protocols represented in the SCRIBBLE protocol description language, and generates protocol-conforming Scala code that uses the EFFPI concurrency library. A user specifies a multiparty protocol in SCRIBBLE as input, introduced in §6.1. We show the style of our generated code in §6.2, and how a developer can use the generated code to implement multiparty protocols. As mentioned in §2, generating channels for each process and type poses an interesting challenge, explained in §6.3.

6.1 Specifying a Multiparty Protocol in Scribble

The SCRIBBLE Language [43] is a multiparty protocol description language that relates closely to MPST theory (cf. [31]), and provides a programmatic way to express global types. As an example, Fig. 10 describes the following global type of a simple distributed logging protocol:

$$G = \mu \mathbf{t}_0. \mathbf{u} \rightarrow \mathbf{l} : \{ \text{write}(\text{str}).\mathbf{t}_0, \text{read}.\mathbf{l} \rightarrow \mathbf{u} : \text{report}(\text{Log}).\mathbf{t}_0, \text{crash}.\text{end} \}.$$

The global type is described by a SCRIBBLE `global protocol`, with roles declared on Line 1. A transmission in the global type (e.g. $\mathbf{u} \rightarrow \mathbf{l} : \{ \dots \}$) is in the form of an interaction statement (e.g. \dots `from U to L`), except that choice (i.e. with an index set $|I| > 1$) must be marked explicitly by a `choice` construct (Line 2). Recursions and type variables in the global types are in the forms of `rec` and `continue` statements, respectively.

In order to express our new theory, we need two extensions to the language:

- (1) a reserved label `crash` to mark crash handling branches (cf. the special label `crash` in the theory), e.g. on Line 7; and
- (2) a `reliable` keyword to mark the reliable roles in the protocol (cf. the reliable role set \mathcal{R} in the theory). Roles are assumed unreliable unless declared using the `reliable` keyword, e.g. L on Line 1.

6.2 Generating Scala Code from Scribble Protocols

The Effpi Concurrency Library. [38] provides an embedded Domain Specific Language (DSL) offering a simple actor-based API. The library utilises advanced type system features in Scala 3, and provides both type-level and value-level constructs for processes and channels. In particular, the type-level constructs reflect the behaviour of programs (i.e. processes), and thus can be used as specifications. Following this intuition, we generate process types that reflect local types from our theory, as well as a tentative process implementing that type (by providing some default values where necessary).

Generated Code. To illustrate our approach, we continue with the Simple Logger example from § 6.1, and show the generated code in Fig. 11. The generated code can be divided into five sections:

- (i) label and payload declarations,
- (ii) recursion variable declarations,
- (iii) local type declarations,
- (iv) role-implementing functions, and
- (v) an entry point.

Sections (i) and (ii) contain boilerplate code, where we generate type declarations for various constructs needed for expressing local types and processes. We draw attention to the *key* sections (iii) and (iv), where we generate a representation of local types for each role, as well as a tentative process inhabiting that type.

Local Types and Effpi Types. We postpone the discussion about channels in EFFPI to § 6.3. For now, we compare the generated EFFPI type and the projected local type, and also give a quick primer³ on EFFPI constructs. The projected local types of the roles **u** and **l** are shown as follows:

$$\begin{aligned} G \upharpoonright_{\{1\}} \mathbf{u} &= \mu t_0. \mathbf{l} \oplus \{ \text{write}(\text{str}).t_0, \text{read}.\mathbf{l} \& \text{report}(\text{Log}).t_0 \} \\ G \upharpoonright_{\{1\}} \mathbf{l} &= \mu t_0. \mathbf{u} \& \{ \text{write}(\text{str}).t_0, \text{read}.\mathbf{u} \oplus \text{report}(\text{Log}).t_0, \text{crash}.\text{end} \} \end{aligned}$$

The local types are recursive, and the EFFPI type implements recursion with `Rec[RecT0, ...]` and `Loop[RecT0]`, using the recursion variable `RecT0` declared in section (ii).

For role **u**, The inner local type is a sending type towards role **l**, and we use an EFFPI process output type `Out[A, B]`, which describes a process that uses a channel of type **A** to send a value of type **B**. For each branch, we use a separate output type, and connect it to the type of the continuation using a sequential composition operator (`>>:`). The different branches are then composed together using a union type (`|`) from the SCALA 3 type system.

Recall that the role **l** is declared `reliable`, and thus the reception labelled `report` from **l** at **u** does not need to contain a crash handler. We use an EFFPI process input type `In[A, B, C]`, which describes a process that uses a channel of type **A** to receive a value of type **B**, and uses the received value in a continuation of type **C**.

For role **l**, the reception type is more complex for two reasons:

- (1) role **u** is unreliable, necessitating crash handling; and
- (2) the reception contains branching behaviour (cf. the reception **u** being a singleton), with labels `write` and `read`.

³ A more detailed description of constructs can be found in [39].

```

1 // (i) label and payload declarations
2 case class Log() // payload type
3 case class Read() // label types
4 case class Report(x : Log)
5 case class Write(x : String)
6 // (ii) recursion variable declarations
7 sealed abstract class RecT0[A]() extends RecVar[A]("RecT0")
8 case object RecT0 extends RecT0[Unit]
9 // (iii) local type declarations
10 type U[C0 <: OutChan[Read | Write], C1 <: InChan[Report]] =
11   Rec[RecT0,
12     ( (Out[C0, Read] >>: In[C1, Report, (x0 : Report) => Loop[RecT0]]
13       | (Out[C0, Write] >>: Loop[RecT0]) ) ]
14
15 type L[C0 <: InChan[Read | Write], C1 <: OutChan[Report]] =
16   Rec[RecT0,
17     InErr[C0, Read | Write, (x0 : Read | Write) => L0[x0.type, C1],
18           (err : Throwable) => PNil]]
19
20 type L0[X0 <: Read | Write, C1 <: OutChan[Report]] <: Process =
21   X0 match { case Read => Out[C1, Report] >>: Loop[RecT0]
22             case Write => Loop[RecT0] }
23 // (iv) role-implementing functions
24 def u(c0 : OutChan[Read | Write],
25     c1 : InChan[Report]) : U[c0.type, c1.type] = {
26   rec(RecT0) {
27     val x0 = 0
28     if (x0 == 0) {
29       send(c0, new Read()) >> receive(c1) {(x1 : Report) => loop(RecT0) }
30     } else {
31       send(c0, new Write("")) >> loop(RecT0)
32     } } }
33
34 def l(c0 : InChan[Read | Write],
35     c1 : OutChan[Report]) : L[c0.type, c1.type] =
36   rec(RecT0) {
37     receiveErr(c0)((x0 : Read | Write) => l0(x0, c1),
38                 (err : Throwable) => nil) }
39
40 def l0(x : Read | Write, c1 : OutChan[Report]) : L0[x.type, c1.type] =
41   x match { case y : Read => send(c1, new Report(new Log())) >> loop(RecT0)
42           case y : Write => loop(RecT0) }
43 // (v) an entry point (main object)
44 object Main {
45   def main() : Unit = {
46     var c0 = Channel[Read | Write]()
47     var c1 = Channel[Report]()
48     eval(par(u(c0, c1), l(c0, c1)))
49   } }

```

■ **Figure 11** Generated SCALA code for the Simple Logger protocol in Fig. 10.

For (1), we extend EFFPI with a variant of the input process type `InErr[A, B, C, D]`, where `D` is the type of continuation in case of a crash. For (2), the payload type is first received as an union (Line 17), and then `matched` to select the correct continuation according to the type (Line 21).

From Types To Implementations. Since EFFPI type-level and value-level constructs are closely related, we can easily generate the processes from the processes types. Namely, by matching the type `Out[...]` with the process `send(...)`; the type `In[...]` with the process `receive(...)` `{... => ...}`; and similarly for other constructs. Whilst executable, the generated code represents a skeleton implementation, and the programmer is expected to alter the code according to their requirements.

We also introduce a new crash handling receive process `receiveErr`, to match the new `InErr` type. Process crashes are modelled by (caught) exceptions and errors in role-implementing functions, and crash detection is achieved via timeouts. Timeouts are set by the programmer in an (implicit) argument to each `receiveErr` call.

Finally, the entry point (main object) in section (v) composes the role-implementing functions together with `par` construct in EFFPI, and connects the processes with channels.

6.3 Generating Effpi Channels from Scribble Protocols

As previously mentioned, EFFPI processes use *channels* to communicate, and the type of the channel is reflected in the type of the process. However, our local types do not have any channels; instead, they contain a partner role with which to communicate. This poses an interesting challenge, and we explain the channel generation procedure in this section.

We draw attention to the generated code in Fig. 11 again, where we now focus on the parameters `C0` in the generated types `U` and `L`. In the type `U`, the channel type `C0` needs to be a subtype of `OutChan[Read | Write]` (Line 10), and we see the channel is used in the output processes types, e.g. `Out[C0, Read]` (Line 12, note that output channels subtyping is covariant on the payload type). Dually, in the type `L`, the channel type `C0` needs to be a subtype of `InChan[Read | Write]` (Line 15), and we see the channel is used in the input process type, i.e. `InErr[C0, Read | Write, ..., ...]` (Line 17).

Similarly, a channel `c0` is needed in the role-implementing functions `u` and `l` as arguments, and the channel is used in processes `send(c0, ...)` and `receiveErr(c0) ...`. Finally, in the entry point, we create a bidirectional channel `c0 = Channel[Read | Write]()` (Line 46), and pass it as an argument to the role-implementing functions `u` and `l` (Line 48), so that the channel can be used to link two role-implementing processes together for communication.

Generating the channels correctly is crucial to the correctness of our approach, but non-trivial since channels are implicit in the protocols. In order to do so, a simple approach is to traverse each interaction in the global protocol, and assign a channel to each accordingly.

This simple approach would work for the example we show in Fig. 10; however, it would not yield the correct result when *merging* occurs during projection, which we explain using an example. For clarity and convenience, we use *annotated* global and local types, where we assign an identifier for each interaction to signify the channel to use, and consider the following global type: $G = p \xrightarrow{0} q: \left\{ \text{left. } p \xrightarrow{1} r: \text{left.end}, \text{ right. } p \xrightarrow{2} r: \text{right.end} \right\}$.

The global type describes a simple protocol, where role `p` selects a label `left` or `right` to `q`, and `q` passes on the same label to `r`. As a result, the projection on `r` (assuming all roles reliable) should be a reception from `q` with branches labelled `left` or `right`, i.e. $p \&^{1,2} \{ \text{left.end}, \text{ right.end} \}$. Here, we notice that the interaction between `q` and `r` should take place on a single channel, instead of two separate channels annotated 1 and 2.

■ **Table 1** Overview of All Variants for Each Example.

Name	Var.	\mathcal{R}	Comms.	Crash Branches	Max Cont. Len.
PingPong $\mathcal{R} = \{p, q\}$	(a)	\mathcal{R}	2	0	4
	(b)	\emptyset	2	2	4
Adder $\mathcal{R} = \{p, q\}$	(c)	\mathcal{R}	5	0	6
	(d)	\emptyset	5	5	6
TwoBuyer $\mathcal{R} = \{p, q, r\}$	(e)	\mathcal{R}	7	0	8
	(f)	$\{r\}$	18	6	12
OAuth $\mathcal{R} = \{c, a, s\}$	(g)	\mathcal{R}	12	0	11
	(h)	$\{s, a\}$	21	8	11
	(i)	$\{s\}$	26	13	11
	(j)	\emptyset	30	28	11
TravelAgency $\mathcal{R} = \{c, a, s\}$	(k)	\mathcal{R}	8	0	6
	(l)	$\{a, s\}$	9	3	6
	(m)	$\{a\}$	9	4	6
DistLogger $\mathcal{R} = \{l, c, i\}$	(n)	\mathcal{R}	10	0	7
	(o)	$\{i, c\}$	15	2	7
	(p)	$\{i\}$	16	4	7
CircBreaker $\mathcal{R} = \{s, a, r\}$	(q)	\mathcal{R}	18	0	10
	(r)	$\{a, s\}$	24	3	10
	(s)	$\{a, s\}$	23	3	11

When *merging* behaviour occurs during projection, we need to use the same channel in those interactions to achieve the correct behaviour. After traversing the global type to annotate each interaction, we merge annotations involved in merges during projection.

7 Evaluation

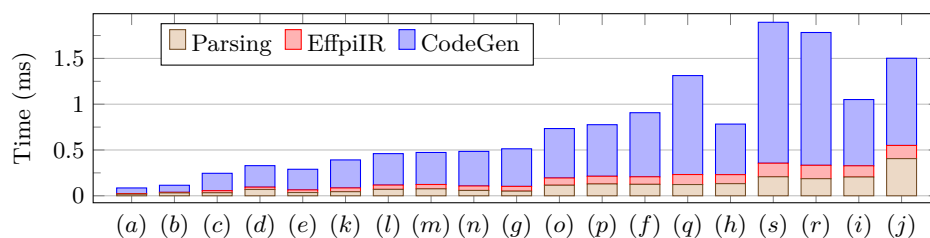
We evaluate our toolchain TEATRINO from two perspectives: *expressivity* and *feasibility*. For expressivity, we use examples from session type literature, and extend them to include crash handling behaviour using two patterns: failover and graceful failure. For feasibility, we show that our tool generates SCALA code within negligible time.

We note that we do *not* evaluate the performance of the generated code. The generated code uses the EFFPI concurrency library to implement protocols, and any performance indication would depend and reflect on the performance of EFFPI, instead of TEATRINO.

Expressivity. We evaluate our approach with examples in session type literature: PingPong, Adder, TwoBuyer [21], OAuth [32], TravelAgency [23], DistLogger [26], and CircBreaker [26]. Notably, the last two are inspired by real-world patterns in distributed computing.

We begin with the *fully reliable* version of the examples, and extend them to include crash handling behaviour. Recall that our extended theory subsumes the original theory, when all roles are assumed *reliable*. Therefore, the fully reliable versions can act both as a sanity check, to ensure the code generation does not exclude good protocols in the original theory, and as a baseline to compare against.

To add crash handling behaviour, we employ two patterns: *failover* and *graceful failure*. In the former scenario, a crashed role has its functions taken over by another role, acting as a substitute to the crashed role [3]. In the latter scenario, the protocol is terminated peacefully, possibly involving additional messages for notification purposes. Using the example from § 2, the fully reliable protocol in Eq. (1) is extended to one with graceful failure in Eq. (2).



■ **Figure 12** Average Generation Times for All Variants in Table 1.

We show a summary of the examples in Table 1. For each example, we give the set of all roles \mathcal{R} and vary the set of reliable roles (\mathcal{R}). Each variant is given an identifier (Var.), and each example always has a fully reliable variant where $\mathcal{R} = \mathcal{R}$. We give the number of communication interactions (Comms.), the number of *crash* branches added (Crash Branches), and the length of the longest continuation (Max Cont. Len.) in the given global type.

The largest of our examples in terms of concrete interactions is *OAuth*, with Variant (*i*) having 26 interactions and (*j*) having 30 interactions. This represents a $2.17\times$ and $2.5\times$ increase over the size of the original protocol, and is a consequence of the confluence of two factors: the graceful failure pattern, and low degree of branching in the protocol itself. The *TwoBuyer* Variant (*f*) represents the greatest increase ($2.57\times$) in interactions, a result of implementing the failover pattern. The *CircBreaker* variants are also notable in that they are large in terms of both interactions and branching degree – both affect generation times.

Feasibility. In order to demonstrate the feasibility of our tool *TEATRINO*, we give generation times using our prototype for all protocol variants and examples, plotted in Fig. 12. We show that *TEATRINO* is able to complete the code generation within milliseconds, which does not pose any major overhead for a developer.

In addition to total generation times, we report measurements for three main constituent phases of *TEATRINO*: parsing, *EffpiIR* generation, and code generation. *EffpiIR* generation projects and transforms a parsed global type into an intermediate representation, which is then used to generate concrete *SCALA* code.

For all variants, the code generation phase is the most expensive phase. This is likely a consequence of traversing the given *EffpiIR* representation of a protocol twice – once for local type declarations and once for role-implementing functions.

8 Related Work

We summarise related work on both theory and implementations of session types with failure handling, as well as other MPST implementations targeting *SCALA* without failures.

We first discuss closest related work [3, 27, 33, 42], where multiparty session types are extended to model crashes or failures. Both [33] and [27] are exclusively theoretical.

[33] proposes an MPST framework to model fine-grained unreliability: each transmission in a global type is parameterised by a reliability annotation, which can be one of unreliable (sender/receiver can crash, and messages can be lost), weakly reliable (sender/receiver can crash, messages are not lost), or reliable (no crashes or message losses). [42] utilises MPST as a guidance for fault-tolerant distributed system with recovery mechanisms. Their framework includes various features, such as sub-sessions, event-driven programming, dynamic role assignments, and, most importantly, failure handling. [3] develops a theory of multiparty

session types with crash-stop failures: they model crash-stop failures in the semantics of processes and session types, where the type system uses a model checker to validate type safety. [27] follow a similar framework to [3]: they model an asynchronous semantics, and support more patterns of failure, including message losses, delays, reordering, as well as link failures and network partitioning. However, their typing system suffers from its genericity, when type-level properties become undecidable [27, §4.4].

Other session type works on modelling failures can be briefly categorised into two: using affine types or exceptions [14, 26, 29], and using coordinators or supervision [1, 41]. The former adapts session types to an *affine* representation, in which endpoints may cease prematurely; the latter, instead, are usually reliant on one or more *reliable* processes that *coordinate* in the event of failure. The works [1, 29, 41] are limited to theory.

[29] first proposes the affine approach to failure handling. Their extension is primarily comprised of a *cancel operator*, which is semantically similar to our crash construct: it represents a process that has terminated early. [14] presents a concurrent λ -calculus based on [29], with asynchronous session-typed communication and exception handling, and implements their approach as parts of the LINKS language. [26] proposes a framework of *affine* multiparty session types, and provides an implementation of affine MPST in the RUST programming language. They utilise the affine type system and `Result` types of RUST, so that the type system enforces that failures are handled.

Coordinator model approaches [1, 41] often incorporate *interrupt blocks* (or similar constructs) to model crashes and failure handling. [1] extends the standard MPST syntax with *optional blocks*, representing regions of a protocol that are susceptible to communication failures. In their approach, if a process P expects a value from an optional block which fails, then a *default value* is provided to P , so P can continue running. This ensures termination and deadlock-freedom. Although this approach does not feature an explicit reliable coordinator process, we describe it here due to the inherent coordination required for multiple processes to start and end an optional block. [41] similarly extends the standard global type syntax with a *try-handle* construct, which is facilitated by the presence of a reliable coordinator process, and via a construct to specify reliable processes. When the coordinator detects a failure, it broadcasts notifications to all remaining live processes; then, the protocol proceeds according to the failure handling continuation specified as part of the try-handle construct.

Other related MPST implementations include [9, 17, 18]. [18] designs a framework for MPST-guided, safe actor programming. Whilst the MPST protocol does not include any failure handling, the actors may fail or raise exceptions, which are handled in a similar way to what we summarise as the affine technique. [9] revisits API generation techniques in SCALA for MPST. In addition to the traditional local type/automata-based code generation [22, 36], they propose a new technique based on sets of pomsets, utilising SCALA 3 match types [4]. [17] presents CHORAL, a programming language for choreographies (multiparty protocols). CHORAL supports the handling of *local exceptions* in choreographies, which can be used to program reliable channels over unreliable networks, supervision mechanisms, *etc.* for fallible communication. They utilise automatic retries to implement channel APIs.

9 Conclusion and Future Work

To overcome the challenge of accounting for failure handling in distributed systems using session types, we propose TEATRINO, a code generation toolchain. It is built on asynchronous MPST with crash-stop semantics, enabling the implementation of multiparty protocols that are resilient to failures. Desirable global type properties such as deadlock-freedom, protocol

conformance, and liveness are preserved by construction in typed processes, even in the presence of crashes. Our toolchain TEATRINO, extends SCRIBBLE and EFFPI to support crash detection and handling, providing developers with a lightweight way to leverage our theory. The evaluation of TEATRINO demonstrates that it can generate SCALA code with minimal overhead, which is made possible by the guarantees provided by our theory.

This work is a new step towards modelling and handling real-world failures using session types, bridging the gap between their theory and applications. As future work, we plan to study different crash models (e.g. crash-recover) and failures of other components (e.g. link failures). These further steps will contribute to our long-term objective of modelling and type-checking well-known consensus algorithms used in large-scale distributed systems.

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 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Designing Asynchronous Multiparty Protocols with Crash-Stop Failures, 2023. arXiv:2305.06238.
- 3 Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. In Bartek Klin, Sławomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR 2022)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:25, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2022.35.
- 4 Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL):1–24, 2022. doi:10.1145/3498698.
- 5 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.
- 6 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.
- 7 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019. doi:10.1145/3290342.
- 8 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.
- 9 Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 27:1–27:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/opus/volltexte/2022/16255>.
- 10 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:10.1145/3503221.3508404.

- 11 Romain Demangeon, Kohei Honda, Raymond Hu, Romyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.
- 12 Romain Demangeon and Nobuko Yoshida. On the Expressiveness of Multiparty Sessions. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*, volume 45 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 560–574, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSTTCS.2015.560.
- 13 Pierre-Malo Deniérou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39212-2_18.
- 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 Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, Alceste Scalas, and Nobuko Yoshida. Precise subtyping for synchronous multiparty sessions. *J. Log. Algebraic Methods Program.*, 104:127–173, 2019. doi:10.1016/j.jlamp.2018.12.002.
- 16 Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434297.
- 17 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. arXiv:2005.09520.
- 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, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems*, pages 122–138, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. doi:10.1007/BFb0053567.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284. ACM, 2008. doi:10.1145/1328897.1328472.
- 21 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63:1–67, 2016. doi:10.1145/2827695.
- 22 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification through Endpoint API Generation. In *19th International Conference on Fundamental Approaches to Software Engineering*, volume 9633 of *LNCS*, pages 401–418, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49665-7_24.
- 23 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 516–541. Springer, 2008. doi:10.1007/978-3-540-70592-5_22.
- 24 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In James Cheney and Germán Vidal, editors, *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, Edinburgh, United Kingdom, September 5-7, 2016*, pages 146–159. ACM, 2016. doi:10.1145/2967973.2968595.

- 25 Nicolas Lagailardie, Romyana Neykova, and Nobuko Yoshida. Implementing Multiparty Session Types in Rust. In Simon Bludze and Laura Bocchi, editors, *Coordination Models and Languages – 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2020. doi:10.1007/978-3-030-50029-0_8.
- 26 Nicolas Lagailardie, Romyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: <https://drops.dagstuhl.de/opus/volltexte/2022/16232>.
- 27 Matthew Alan Le Brun and Ornela Dardha. $MAG\pi$: Types for Failure-Prone Communication. In Thomas Wies, editor, *Programming Languages and Systems*, pages 363–391, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30044-8_14.
- 28 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-Safe Web Programming in TypeScript with Routed Multiparty Session Types. In *International Conference on Compiler Construction*, CC, pages 94–106, 2021. doi:10.1145/3446804.3446854.
- 29 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.
- 30 Romyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *Logical Methods in Computer Science*, 13:1–30, 2017. doi:10.23638/LMCS-13(1:17)2017.
- 31 Romyana Neykova and Nobuko Yoshida. *Featherweight Scribble*, pages 236–259. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-21485-2_14.
- 32 Romyana Neykova, Nobuko Yoshida, and Raymond Hu. SPY: Local Verification of Global Protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40787-1_25.
- 33 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*, pages 93–113, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-08679-3_7.
- 34 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
- 35 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011. doi:10.1017/CB09780511777110.
- 36 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:31, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2017.24.
- 37 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.
- 38 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, pages 502–516, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3322484.
- 39 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Effpi: verified message-passing programs in Dotty. In Jonathan Immanuel Brachthäuser, Sukyoung Ryu, and Nathaniel Nystrom, editors, *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala@ECOOP 2019, London, UK, July 17, 2019*, pages 27–31. ACM, 2019. doi:10.1145/3337932.3338812.
- 40 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.

1:30 Designing Asynchronous Multiparty Protocols with Crash-Stop Failures

- 41 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. doi:10.1007/978-3-319-89884-1_28.
- 42 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), October 2021. doi:10.1145/3485501.
- 43 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *8th International Symposium on Trustworthy Global Computing – Volume 8358*, TGC 2013, pages 22–41, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-319-05119-2_3.