Model Checking Probabilistic Multiparty Session Types

ALEKSANDER JUNGE* and LEO SEOJUN LEE*, University of Oxford, UK
JOSEPH PAULUS, NOBUKO YOSHIDA, and DAVID PARKER, University of Oxford, UK

Distributed systems increasingly rely on randomisation; yet, verification techniques for such systems remain separate from the programming abstractions used to build them. This paper proposes the first probabilistic extension of *general multiparty session types* — typing sessions bottom-up, with no global specification — deriving quantitative properties of a multiparty session from its session types alone. Our types combine *probabilistic* choice with *nondeterministic* choice that abstracts data-dependent control, and allow *separated choices*, where each branch of a selection or branching may involve a different participant — a combination beyond previous probabilistic multiparty session type systems. To achieve this, we develop a novel probabilistic subtyping relation and merge operator, using which we prove type soundness and session fidelity. The semantics of typing contexts separates nondeterministic resolution, probabilistic choice, and synchronisation, so that every typing context induces a finite Markov decision process. On it, we derive sound *lower bounds*, holding under every fair adversarial scheduler, on the probability that a session is deadlock-free and live — the first behavioural property guarantees, partial or otherwise, for a probabilistic session calculus. To connect session types with probabilistic model checking, we introduce the first translation from session types into the PRISM modelling language, and prove it sound and complete with respect to our operational semantics. We implement the framework in the toolchain Prose, which translates probabilistic multiparty session typing contexts into PRISM and checks safety, deadlock-freedom, and liveness.

CCS Concepts: • **Theory of computation → Verification by model checking**; *Probabilistic computation*; *Process calculi*; **Type theory**.

Additional Key Words and Phrases: multiparty session types, probabilistic processes, probabilistic model checking, type systems, safety, probabilistic deadlock-freedom and liveness.

ACM Reference Format:
Aleksander Junge, Leo Seojun Lee, Joseph Paulus, Nobuko Yoshida, and David Parker. 2026. Model Checking Probabilistic Multiparty Session Types. 1, 1 (November 2026), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

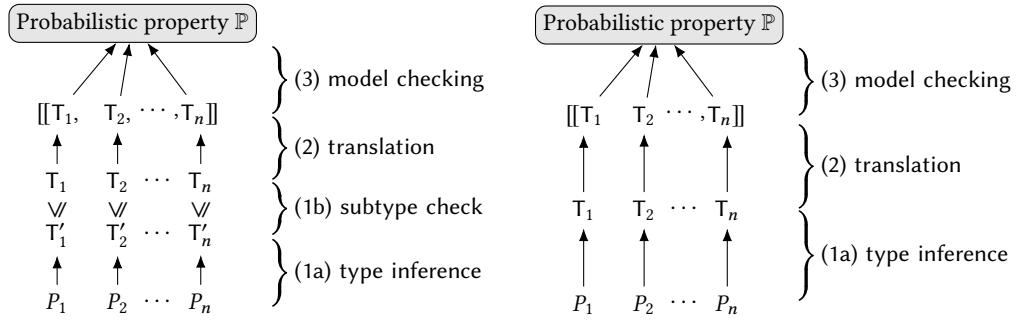
1 Introduction

Probabilistic model checking. Concurrent components have been deployed in many computing systems, from multicore architectures to web services to autonomous systems. Ensuring their safe and reliable execution is an essential requirement for the design and specification of system components, but it creates additional demand for verification, as components operate in *uncertain* environments. *Probabilistic model checking* [5, 6] aims to automatically check whether components that exhibit uncertain (probabilistic) and nondeterministic behaviours satisfy given requirements,

*Both authors contributed equally to this research.

Authors' Contact Information: Aleksander Junge, aleksander.junge@cs.ox.ac.uk; Leo Seojun Lee, leo@leol.ee, University of Oxford, Oxford, UK; Joseph Paulus, joe.paulus@cs.ox.ac.uk; Nobuko Yoshida, nobuko.yoshida@cs.ox.ac.uk; David Parker, david.parker@cs.ox.ac.uk, University of Oxford, Oxford, UK.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Type checking (b) Type Inference

Fig. 1. Overview of the probabilistic property verification procedure. In (a), given T_i and P_i ($i \in \{1..n\}$) (1a) infer local type T'_i from process P_i; (1b) check $\mathsf{T}'_i \leqslant \mathsf{T}_i$; (2) translate T_i to PRISM and (3) model check whether $\{T_i\}_{i\in I}$ satisfy the desired probabilistic property $\mathbb{P}$. In (b), directly verify the inferred types.

usually specified by probabilistic extensions of temporal logics. Probability is used to quantify environmental uncertainty or randomisation, nondeterminism to represent decisions or concurrency. Over the last two decades, probabilistic model checking tools (e.g., [3, 25, 26, 38]) have been actively developed and used for quantitative verification in a broad spectrum of application domains [39]. Our aim is to lay the first groundwork connecting *probabilistic multiparty session types* to probabilistic model checking, so that session-typed systems benefit directly from advances in model checking techniques. Through a *precise* behavioural correspondence between probabilistic session types and a model checker's input language, we analyse a spectrum of probabilistic properties of multiparty sessions, developing a static typing system that yields sound bounds on the probability of "good" behaviour — such as deadlock-freedom and liveness.

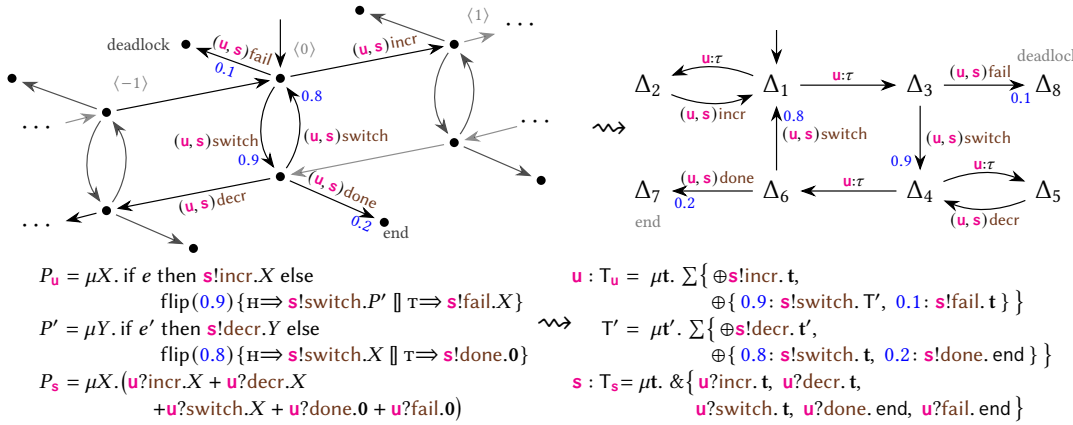
Multiparty Session Types (MPST) [30, 31] are an extension of binary session types [29, 55], facilitating specification and verification of complex many-party interacting components. Historically, the MPST theory has been developed based on the *top-down approach* where the chief system architect first writes a multiparty protocol (called a *global type*) between participants. This protocol is projected to a set of end-point (*local*) types against which conformance of each component (participant) is checked. A collection of typed components then automatically satisfies *qualitative properties* such as type-safety, deadlock-freedom and liveness [64].

More recently, Scalas and Yoshida [52] proposed *generalised multiparty session types* that do *not* require global types, but can enforce a desired property φ by directly checking that a set of local types satisfies φ. The bottom-up approach types a *strictly larger set* of processes: protocols beyond the reach of any global type — with asymmetric or emergent interaction structures that projection cannot express — as well as type-safe processes that may deadlock, run forever, or fail to progress. This "partial" nature is a natural fit with probabilistic model checking: rather than demanding the outright absence of deadlock or non-termination, tools such as PRISM and Storm *quantify* such behaviours — e.g., computing the probability of reaching a deadlock or of termination [3, 28].

Overview

We give an overview of the workflow in Fig. 1 and explain each step below.

Step (1): Type checking. Following the bottom-up approach [52], the first stage of our framework abstracts each process as a finite probabilistic session type T. This abstraction is necessary: communication safety of untyped session processes is undecidable, as we prove by encoding



$P_u = \mu X.$ if e then s!incr.X else flip(0.9){H⟹ s!switch.P' [] T⟹ s!fail.X}

$P' = \mu Y.$ if e' then s!decr.Y else flip(0.8){H⟹ s!switch.X [] T⟹ s!done.**0**}

$P_s = \mu X.$(u?incr.X + u?decr.X +u?switch.X + u?done.**0** + u?fail.**0**)

u : $\mathsf{T}_u = \mu$t. ∑{ ⊕s!incr. t, ⊕{ 0.9: s!switch. T′, 0.1: s!fail. t } }

$\mathsf{T}' = \mu$t′. ∑{ ⊕s!decr. t′, ⊕{ 0.8: s!switch. t, 0.2: s!done. end } }

s : $\mathsf{T}_s = \mu$t. &{ u?incr. t, u?decr. t, u?switch. t, u?done. end, u?fail. end }

Fig. 2. An infinite-state session (left; one column per counter value) and its finite typing context Δ_i (right). The data-dependent decision by if becomes a nondeterministic choice over its two alternatives, while the probabilities of flip are preserved as edge annotations. Processes are shown bottom-left and their local types bottom-right. "μ" denotes recursion, !/? output/input, and ⊕/& internal/external choice (see §2 and 3).

two-counter Minsky machines into sessions (Prop. 2.3). After inferring local types our toolchain supports two modes; either model check the types directly (Fig. 1b), or type check them against a type specification via subtyping before model checking (Fig. 1a).

Fig. 2 shows a session (bottom-left) in which a worker **u** drives a counter held by **s** up or down, switching mode depending on its state. Following [20], our calculus distinguishes data-dependent conditionals (if) from probabilistic choices (flip). Switching down fails with probability 0.1, resulting in a deadlock, while switching up terminates successfully with probability 0.2.

The unbounded counter gives rise to an infinite state space (left, with one column per counter value), preventing direct finite-state model checking. We instead infer local types that soundly over-approximate communication behaviour, following existing approaches that model check behavioural types as *abstractions of processes* [52] and *programs* [7, 42, 53].

Typing replaces data-dependent conditionals with nondeterministic choices, accounting for the possible outcomes of tests whose values the types do not track, while retaining probability information from flips. In Fig. 2, this removes the need to track the unbounded counter: the types describe a finite model of the communication modes instead.

The resulting model is a *finite Markov decision process* (MDP), combining probabilistic choices with nondeterminism arising from both abstracted data and participant scheduling. We analyse it under the least favourable (adversarial) resolutions of nondeterminism, subject to our fairness assumptions. This accounts for the information lost through abstraction and yields *sound lower bounds* on the probabilities of properties of the original session. Type checking is decidable (Thm. 3.14), and the finite abstraction enables probabilistic model checking. For instance, this analysis establishes that the session in Fig. 2 is deadlock-free with probability at least $9/14 \approx 0.64$ under every fair scheduler.

Step (2): Precise translation from session types into PRISM. To verify type-level properties automatically, we employ the PRISM [38] probabilistic model checker, a well-established tool which has been used to study a diverse range of applications [39]. Other popular probabilistic model checkers such as Storm [58] and the Modest Toolset [25] also support the PRISM language as input (the latter by converting first into the JANI [11] format).

To fill a gap between types and models, we *encode* session types into the PRISM modelling language and prove a *precise* (*sound* and *complete*) behavioural correspondence. Building on earlier

approaches to model checking behavioural types [7, 42, 52, 53], we verify the translation itself by developing new bisimulation and operational correspondence relations for the PRISM semantics, precisely relating type-level behaviours to those of the generated models.

Step (3): Probabilistic property analysis and model checking. As the final step, we express the properties of typing contexts (§4.3) as specifications in Probabilistic Linear Temporal Logic (PLTL) and run PRISM to verify them. For deadlock-freedom and liveness this establishes lower bounds (§4.3), while for safety we get a yes/no answer.

There are two main challenges, both rooted in the nondeterminism of the type-level MDP. First, the probability of a set of paths is well-defined only relative to a *scheduler*: both the resolution of the abstracted data-dependent choices and the scheduling of parallel roles can change the probability of satisfying a given property. We define the probability of a set of paths under a fixed scheduler by standard means, using *cylinder sets* (Def. 4.7), and obtain lower bounds by taking the *minimum* over all fair schedulers (Def. 4.6); these minima then transfer to well-typed sessions (Theorems 4.26 and 4.27). The second challenge is that *probabilistic deadlock-freedom* and *probabilistic liveness* are quantified over *fair* schedulers only, while model checkers impose fairness on all nondeterministic choices uniformly—including the internal nondeterminism, which must remain unconstrained. We therefore characterise probabilistic deadlock-freedom as a safety property over finite paths that PRISM checks without any fairness constraint. Characterising liveness in this manner is more challenging: to do this, we introduce the notion of a *weak almost-sure livelocked* region, and prove that the maximum probability of reaching this region corresponds exactly to the minimum probability that the typing context is live (Theorem 6.5).

Contributions and Outline. To the best of our knowledge, this work is the first to propose a probabilistic extension of generalised (bottom-up) multiparty session types, to prove that a typing context yields *sound lower bounds* of probabilistic *properties* of typed session processes, and to translate session types into a probabilistic modelling language with proofs of *soundness* and *completeness* with respect to operational semantics. **§2** proposes a *probabilistic multiparty session calculus* (ProMT) as a conservative extension of synchronous multiparty session processes in [22], with a single probabilistic construct $\mathsf{flip}(p)\{\mathrm{H} \Rightarrow P \parallel \mathrm{T} \Rightarrow Q\}$, and proves that safety of sessions is undecidable (Proposition 2.3). **§3** proposes a *probabilistic multiparty session typing system*, introducing *probabilistic subtyping* (Def. 3.2), which widens data-dependent choices into nondeterministic sums, and the *probabilistic merging operator* (Def. 3.8), which reconciles the two outcomes of a flip. We further demonstrate that type checking is decidable (Theorem 3.14). **§4** proves *subject reduction (type safety)* (Theorem 4.1) and *session fidelity* (Theorem 4.4), interprets typing context reductions as finite MDPs with path probabilities relative to schedulers (Def. 4.6), and proves the main results: a typing context induces *lower bounds* on the probabilities of deadlock-freedom (Theorem 4.26) and liveness (Theorem 4.27) of the sessions it types, with their non-probabilistic counterparts as special cases. **§5** proposes a translation from session types to the PRISM language and proves sound and complete operational correspondence (Theorems 5.6 and 5.7). **§6** expresses safety, deadlock-freedom and liveness as PLTL specifications, reducing liveness to reachability of weak almost-sure livelocks (Theorem 6.5), benchmarks our toolchain Prose on case studies from the PRISM homepage and related literature, and outlines a prototype implementation of steps (1a) and (1b) of Fig. 1. **§7** concludes with related work. Detailed proofs and additional examples are available in the full version of this paper [36]. Our artifact is explained in §8 and available in [35].

2 Probabilistic Multiparty Processes and Sessions

This section introduces the syntax and semantics of a *probabilistic multiparty session calculus* (ProMT-calculus). Since our focus is *probabilistic verification*, we base our calculus on the simplest multiparty

session calculus [22] extended with a branching construct (external choice) which receives a message from different participants [51] and add a single new construct, $\mathsf{flip}(p)\{\text{H}\Rightarrow P \,[\!]\, \text{T}\Rightarrow Q\}$.

Probability and nondeterminism. The calculus separates two sources of branching behaviour. *Probability* is introduced operationally by $\mathsf{flip}(p)\{\text{H}\Rightarrow P \,[\!]\, \text{T}\Rightarrow Q\}$, whose two continuations are taken with complementary probabilities p and $1-p$; the reduction relation tracks these as explicit weights. *Nondeterminism* arises at the process level from the scheduling of parallel roles. The typed account of how nondeterminism is abstracted is deferred to §3.

Expressions $(\mathsf{e}, \mathsf{e}', \dots)$ are variables $(x, y, z, \dots)$, a value v, which is either a natural number n, an integer i, or a Boolean true/false, or expressions built by applying operators such as neg, $\neg$ and $+$.

$$\mathsf{e} ::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{n} \mid \mathsf{i} \mid x \mid \mathsf{e} \vee \mathsf{e} \mid \neg\mathsf{e} \mid \mathsf{e} + \mathsf{e} \mid \mathsf{succ}(\mathsf{e}) \mid \mathsf{neg}(\mathsf{e}) \mid \mathsf{e} > \mathsf{e}'$$

Processes $(P, Q, R, \dots)$ and **multiparty sessions** $(M, M', \dots)$ are defined by the following syntax:

$$\begin{array}{rll} P, Q ::= & \mathbf{0} \mid \mathsf{p}!\ell\langle\mathsf{e}\rangle.P \mid \Sigma_{i\in I}\mathsf{p_i}?\ell_i(x_i).P_i \mid \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q \mid & \text{(nil, selection, branching, conditional)} \\ & \mathsf{flip}(p)\{\text{H}\Rightarrow P \,[\!]\, \text{T}\Rightarrow Q\} \mid \mu X.P \mid X & \text{(flip, recursion, proc variable)} \\ M, M' ::= & \mathsf{p} \triangleleft P \mid M \mid M' \mid \mathsf{error} & \text{(role, parallel composition, error)} \end{array}$$

$\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \dots$ are *participants* (*roles*) from a countably infinite set. The *nil* $\mathbf{0}$ denotes termination; process $\mathsf{p}!\ell\langle\mathsf{e}\rangle.P$ denotes the *selection* of label ℓ and expression e communicating with participant p and continuing with behaviour P. Process $\Sigma_{i\in I}\mathsf{p_i}?\ell_i(x_i).P_i$ is *branching* (*external choice*): for any $i \in I \neq \emptyset$ it can accept label ℓ_i with a value bound to x_i from participant $\mathsf{p_i}$, continuing as P_i. The sender $\mathsf{p_i}$ may vary across branches, allowing a single external choice to await messages from different participants. This follows the external choice in [50, 51], which can represent protocols in which a server accepts client updates in arbitrary order, as in session-typed models of federated learning. On the sending side, as $\mathsf{p}!\ell\langle\mathsf{e}\rangle.P$ names its own target p, a role may also direct the branches of a flip or if statement to *different* participants. We often write $\mathsf{p}?\ell(x).P$ for a singleton branching, expand it as, e.g., $\mathsf{p_1}?\ell_1(x_1).P_1 + \mathsf{p_2}?\ell_2(x_2).P_2$, and omit the payloads, e.g., $\mathsf{p_1}?\ell_1.P_1 + \mathsf{p_2}?\ell_2.P_2$, when they are not needed. *Conditional processes* $\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and *recursive processes* $\mu X.P$ are standard. We assume recursive processes are *guarded*, i.e., every recursive variable occurs under a prefix containing at least one input or output. Thus e.g. $\mu X.X$ and $\mu X.\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ X\ \mathsf{else}\ Q$ are not valid, whereas $\mu X.\mathsf{p}?\ell(x).X$ is. We let p, p', p'' range over *probabilities* and introduce the flip construct $\mathsf{flip}(p)\{\text{H}\Rightarrow P \,[\!]\, \text{T}\Rightarrow Q\}$ which continues as P on heads and Q on tails; we syntactically constrain $p \in (0, 1)$ to the *open* unit interval, so that neither outcome of a flip is excluded.

Process P plays the participant p (denoted $\mathsf{p} \triangleleft P$) and can interact with other roles through parallel composition $\mid$, which we assume to be commutative and associative; we write $\prod_{i\in I} \mathsf{p}_i \triangleleft P_i$ for $\mathsf{p_1} \triangleleft P_1 \mid \cdots \mid \mathsf{p}_n \triangleleft P_n$ with $I = \{1, \dots, n\}$. error denotes an error session.

Remark 2.1 (PROMT probability constructs)**.** The probabilistic behaviour of PROMT only originates from $\mathsf{flip}(p)\{\text{H}\Rightarrow P \,[\!]\, \text{T}\Rightarrow Q\}$, inspired by the flip of [17] and close to the top-level probabilistic internal choice $P \boxplus_p Q$ of [34]. PROMT has no primitive probabilistic output choice as in [1, 2, 60], but simulates one:

$$P \;=\; \mathsf{flip}(p)\{\text{H}\Rightarrow \mathsf{p_1}!\ell_1\langle\mathsf{e}_1\rangle.P_1 \,[\!]\, \text{T}\Rightarrow \mathsf{p_2}!\ell_2\langle\mathsf{e}_2\rangle.P_2\}$$

emits $\ell_1\langle\mathsf{e}_1\rangle$ on heads (probability p) and $\ell_2\langle\mathsf{e}_2\rangle$ on tails $(1-p)$, which in [1, 2, 60] would be written as the explicit probabilistic output choice $\mathsf{p_1}!\,p{:}\ell_1\langle\mathsf{e}_1\rangle.P_1 \;\boxplus_p\; \mathsf{p_2}!\,(1-p){:}\ell_2\langle\mathsf{e}_2\rangle.P_2$. Nesting flips extends this construction to probabilistic output choices over any finite number of alternatives.

Example 2.1 (Calculator service). A user u submits integers to a server s, which forwards the request to a calculator c; the calculator may (depending on the size of the integer) occasionally produce an error. The three roles realise the session $M = \mathsf{u} \triangleleft P_\mathsf{u} \mid \mathsf{s} \triangleleft P_\mathsf{s} \mid \mathsf{c} \triangleleft P_\mathsf{c}$ where $P_\mathsf{s}, P_\mathsf{c}, P_\mathsf{u}$ are:

$$[\text{R-HEAD}]\quad \mathsf{p} \triangleleft \mathsf{flip}(p)\{\text{H}\Rightarrow P \parallel \text{T}\Rightarrow Q\} \mid M \to_p \mathsf{p} \triangleleft P \mid M \qquad [\text{R-TAIL}]\quad \mathsf{p} \triangleleft \mathsf{flip}(p)\{\text{H}\Rightarrow P \parallel \text{T}\Rightarrow Q\} \mid M \to_{1-p} \mathsf{p} \triangleleft Q \mid M$$

$$[\text{R-T-IF}]\ \frac{\mathsf{e} \downarrow \mathsf{true}}{\mathsf{p} \triangleleft \mathsf{if\ e\ then}\ P\ \mathsf{else}\ Q \mid M \to_1 \mathsf{p} \triangleleft P \mid M} \qquad [\text{R-F-IF}]\ \frac{\mathsf{e} \downarrow \mathsf{false}}{\mathsf{p} \triangleleft \mathsf{if\ e\ then}\ P\ \mathsf{else}\ Q \mid M \to_1 \mathsf{p} \triangleleft Q \mid M}$$

$$[\text{R-COMM}]\ \frac{j \in I \qquad \ell = \ell_j \qquad \mathsf{p} = \mathsf{p}_j \qquad \mathsf{e} \downarrow \mathsf{v}}{\mathsf{p} \triangleleft \mathsf{q}!\ell\langle \mathsf{e}\rangle.P \mid \mathsf{q} \triangleleft \textstyle\sum_{i\in I} \mathsf{p}_i?\ell_i(x_i).Q_i \mid M \to_1 \mathsf{p} \triangleleft P \mid \mathsf{q} \triangleleft Q_j\{\mathsf{v}/x_j\} \mid M}$$

$$[\text{R-ERR}]\ \frac{\exists j \in I : \mathsf{p} = \mathsf{p}_j, \quad \nexists i \in I : \mathsf{p}_i = \mathsf{p} \wedge \ell_i = \ell}{\mathsf{p} \triangleleft \mathsf{q}!\ell\langle \mathsf{e}\rangle.P \mid \mathsf{q} \triangleleft \textstyle\sum_{i\in I} \mathsf{p}_i?\ell_i(x_i).Q_i \mid M \longrightarrow_1 \mathsf{error}} \qquad [\text{R-SC}]\ \frac{M_1 \Rrightarrow M_1' \quad M_1' \to_p M_2 \quad M_2 \Rrightarrow M_2'}{M_1 \to_p M_2'}$$

Fig. 3. Reduction rules for PROMT-calculus

$$P_{\mathsf{u}} = \mathsf{s}!\mathsf{incr}\langle 1\rangle.\mu X.\, \mathsf{c}?\mathsf{res}(x).\mathsf{s}!\mathsf{incr}\langle x\rangle.X \qquad P_{\mathsf{s}} = \mu X.\big(\mathsf{c}?\mathsf{error}(x).\mathsf{c}!\mathsf{incr}\langle x\rangle.X \ +\ \mathsf{u}?\mathsf{incr}(x).\mathsf{c}!\mathsf{incr}\langle x\rangle.X\big)$$

$$P_{\mathsf{c}} = \mu X.\mathsf{s}?\mathsf{incr}(x).\mathsf{if}\ x < 2^8\ \mathsf{then}\ \mathsf{u}!\mathsf{res}\langle \mathsf{succ}(x)\rangle.X\ \mathsf{else}\ \mathsf{flip}(0.8)\{\text{H}\Rightarrow \mathsf{u}!\mathsf{res}\langle \mathsf{succ}(x)\rangle.X \parallel \text{T}\Rightarrow \mathsf{s}!\mathsf{error}\langle x\rangle.X\}$$

The user loops, sending $\mathsf{incr}\langle e\rangle$ and then awaits a result $\mathsf{res}(x)$ from c, binding the result of the computation to x before recursing. The server waits for one of two external choices: (1) incr from u, after which it sends $\mathsf{incr}\langle \mathsf{e}\rangle$ to c and loops; or (2) an $\mathsf{error}(x)$ from the calculator c on a previous computation upon which it retries the calculation by sending $\mathsf{incr}\langle x\rangle$ to c again. The calculator is accurate up to a certain integer size — if the input from s is smaller than 2^8, c successfully increments x and forwards it to u, on the other hand if the number is greater than or equal, it has a 0.2 probability of sending error to s before recursing (and 0.8 probability of succeeding as before).

Reduction relations. We write $\mathsf{e} \downarrow \mathsf{v}$ when fully evaluating a closed expression e (one with no free variables) yields value v; the rules are standard [22, Table 1]. The reduction relation for sessions is given in Fig. 3; reductions $\longrightarrow_p$ are annotated with a weight that tracks probabilistic reduction paths. The probabilistic rules are [R-HEAD] and [R-TAIL], taken when a $\mathsf{flip}(p)\{\text{H}\Rightarrow P \parallel \text{T}\Rightarrow Q\}$ resolves—heads (weight p) continuing as P, tails ($1-p$) as Q; all other rules carry weight 1 and coincide with the non-probabilistic MPST calculus of [22, 59]. Rules [R-T-IF] and [R-F-IF] fire once the guard e has evaluated to true or false. In [R-COMM], a participant p selecting label ℓ to q synchronises with q's matching branch (the one whose sender is p), binding v to x_j in the continuation Q_j; the weight is 1, as the communication surely happens. Rule [R-ERR] is enabled when p and q can communicate but there's a label *mismatch*, i.e. p sends a label ℓ not expected by q. Rule [R-SC] closes reduction under the *structural precongruence* $\Rrightarrow$, defined by $\mathsf{p} \triangleleft \mu X.P \mid M \Rrightarrow \mathsf{p} \triangleleft P\{\mu X.P/X\} \mid M$ and $\prod_{i\in I} \mathsf{p}_i \triangleleft P_i \Rrightarrow \prod_{j\in J} \mathsf{p}_j \triangleleft P_j$ when I is a permutation of J; we write $\equiv$ for $(\Rrightarrow \cup \Rrightarrow^{-1})$. We use $\to_p^*$ for the *reflexive* and *transitive closure* of $\longrightarrow_{p'}$, where p is the cumulative probability of that path, e.g. if $M \longrightarrow_{p'} \longrightarrow_{p''} M'$ then $M \to^*_{p'\cdot p''} M'$.

Example 2.2 (Reductions). We trace a reduction of the session M from Ex. 2.1. We abbreviate $P'_{\mathsf{u}} = \mu X.\mathsf{c}?\mathsf{res}(x).\mathsf{s}!\mathsf{incr}\langle x\rangle.X$ for the loop of u, and unfold recursion silently by [R-SC]:

$$\begin{aligned}
&\xrightarrow[1]{\text{[R-COMM]}} && \mathsf{u} \triangleleft \mathsf{c}?\mathsf{res}(x).\mathsf{s}!\mathsf{incr}\langle x\rangle.P'_{\mathsf{u}} \mid \mathsf{s} \triangleleft \mathsf{c}!\mathsf{incr}\langle 1\rangle.P_{\mathsf{s}} \mid \mathsf{c} \triangleleft P_{\mathsf{c}} && (\mathsf{u} \xrightarrow{\mathsf{incr}\langle 1\rangle} \mathsf{s})\\
&\xrightarrow[1]{\text{[R-COMM]}} && \mathsf{u} \triangleleft \mathsf{c}?\mathsf{res}(x).\mathsf{s}!\mathsf{incr}\langle x\rangle.P'_{\mathsf{u}} \mid \mathsf{s} \triangleleft P_{\mathsf{s}} \mid \mathsf{c} \triangleleft \mathsf{if}\ 1 < 2^8\ \mathsf{then}\ \mathsf{u}!\mathsf{res}\langle 2\rangle.P_{\mathsf{c}}\ \mathsf{else}\ \ldots && (\mathsf{s} \xrightarrow{\mathsf{incr}\langle 1\rangle} \mathsf{c})\\
&\xrightarrow[1]{\text{[R-T-IF]}} && \mathsf{u} \triangleleft \mathsf{c}?\mathsf{res}(x).\mathsf{s}!\mathsf{incr}\langle x\rangle.P'_{\mathsf{u}} \mid \mathsf{s} \triangleleft P_{\mathsf{s}} \mid \mathsf{c} \triangleleft \mathsf{u}!\mathsf{res}\langle 2\rangle.P_{\mathsf{c}} && (\text{since } 1 < 2^8)\\
&\xrightarrow[1]{\text{[R-COMM]}} && \mathsf{u} \triangleleft \mathsf{s}!\mathsf{incr}\langle 2\rangle.P'_{\mathsf{u}} \mid \mathsf{s} \triangleleft P_{\mathsf{s}} \mid \mathsf{c} \triangleleft P_{\mathsf{c}} && (\mathsf{c} \xrightarrow{\mathsf{res}\langle 2\rangle} \mathsf{u})
\end{aligned}$$

and the value loops back incremented. For larger inputs the calculator may error and the edges would carry a probability $p \neq 1$.

However, our processes are too expressive to analyse directly. Indeed, the following result tells us that safety of (untyped) processes cannot be established in a sound and complete manner.

Proposition 2.3. *It is undecidable to check safety of (untyped) sessions: no algorithm exists that, given an arbitrary session M, can decide whether $M \not\longmapsto^* \mathsf{error}$.*

Proof. We reduce the halting problem to the problem under consideration. A two-counter Minsky machine $\mathcal{M}$ is encoded as a session $\langle\!\langle \mathcal{M} \rangle\!\rangle$ that simulates it and reaches error exactly when $\mathcal{M}$ reaches its HALT instruction; hence $\langle\!\langle \mathcal{M} \rangle\!\rangle$ is safe iff $\mathcal{M}$ does not halt. As halting is undecidable [47], so is safety. Full proof in [36, App. A.1]. □

This motivates a type system for promt: typing provides a *decidable*, *sound* criterion for safety (well-typed ⇒ safe), and safety alone forces incompleteness (safe ⇏ well-typed), irrespective of any further properties (deadlock-freedom and liveness) we wish to analyse. The abstraction by types allows us to check safety and other probabilistic properties in a tractable manner.

3 Probabilistic Multiparty Session Types and Typing System

3.1 Probabilistic Multiparty Session Types

We start by defining probabilistic multiparty session types, subtyping and merging. We then define their labelled transition relation, and properties of typing contexts. Proofs of statements in this section can be found in the full version [36, App. B.3].

Probabilistic Multiparty Session Types $(\mathsf{T}, \mathsf{T}', \ldots)$ define a probabilistic communication specification for each participant. *Sorts* (or *base types*) are ranged over by $\mathsf{B}, \mathsf{B}', \ldots$

$$\mathsf{B} ::= \mathtt{bool} \mid \mathtt{nat} \mid \mathtt{int} \qquad \mathsf{T} ::= \mathsf{end} \mid \textstyle\sum_{i\in I} \bigoplus_{j\in J_i} p_{i,j} : \mathtt{p_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j} \mid \&_{i\in I}\, \mathtt{p_i}?\ell_i(\mathsf{B}_i).\mathsf{T}_i \mid \mu \mathbf{t}.\mathsf{T} \mid \mathbf{t}$$

Sorts are boolean, naturals and integers; *probabilistic multiparty session types* are: type end denoting termination; type $\&_{i\in I}\ \mathtt{p_i}?\ell_i(\mathsf{B}_i).\mathsf{T}_i$ denoting *branching* (external choice) for each i offering label ℓ_i to $\mathtt{p_i}$. For *branching* to be well-formed, we require each $\mathtt{p_i}?\ell_i$ be distinct and $I \neq \emptyset$. Type $\sum_{i\in I}\bigoplus_{j\in J_i} p_{i,j} : \mathtt{p_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j}$ denotes *selection* (internal choice) as a two-step resolution; the outer sum $\sum_{i\in I}$ represents a *non-deterministic* choice over its summands $\bigoplus_{j\in J_i} p_{i,j} : \mathtt{p_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j}$, that each denote a discrete output distribution with a fixed probability $p_{i,j} \in (0, 1]$ assigned to outputting $\ell_j\langle \mathsf{B}_j\rangle$ to $\mathtt{p_j}$ and continuing as $\mathsf{T}_{i,j}$. For *selection* to be *well-formed* we require $I \neq \emptyset$, each $\mathtt{p_j}!\ell_j$ $j \in J_i$ be distinct and $\sum_{j\in J_i} p_{i,j} = 1$ for all i. We sometimes write $\bigoplus_{j\in J_i} p_{1,j} : \mathtt{p_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{1,j} + \sum_{i\in I\setminus\{1\}}\bigoplus_{j\in J_i} p_{i,j} : \mathtt{p_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j}$ for $\sum_{i\in I}\bigoplus_{j\in J_i} p_{i,j} : \mathtt{p_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j}$ and often omit $\sum_{i\in I}$ when $|I| = 1$. $\mu\mathbf{t}.\mathsf{T}$ and $\mathbf{t}$ are *recursive types* and *type variables*, where T is *guarded*: every type variable occurs under a branching or selection prefix (so $\mu\mathbf{t}.\mathbf{t}$ is excluded), ensuring contractiveness. We often omit end, base types B and the probability 1. We denote $\mathtt{ftv}(\mathsf{T})$ as the set of *free type variables* in T, and call T *closed* if $\mathtt{ftv}(\mathsf{T}) = \emptyset$. Hereafter we assume all types are closed and well-formed.

Example 3.1 (Calculator service continued). Consider again the *user-server-calculator* session from Example 2.1. Letting $\mathtt{p} : \mathsf{T}$ denote that participant $\mathtt{p}$ has type T, the session is typed as:

$$\mathtt{u} : \mathsf{T}_\mathtt{u} = \mu\mathbf{t}.\ \oplus\, \mathtt{s}!\mathtt{incr}\langle\mathtt{int}\rangle.\, \&\mathtt{c}?\mathtt{res}(\mathtt{int}).\, \mathbf{t}$$

$$\mathtt{s} : \mathsf{T}_\mathtt{s} = \mu\mathbf{t}.\ \&\big\{\, \mathtt{c}?\mathtt{error}(\mathtt{int}).\, \oplus\, \mathtt{c}!\mathtt{incr}\langle\mathtt{int}\rangle.\, \mathbf{t},\ \ \mathtt{u}?\mathtt{incr}(\mathtt{int}).\, \oplus\, \mathtt{c}!\mathtt{incr}\langle\mathtt{int}\rangle.\, \mathbf{t}\,\big\}$$

$$\mathtt{c} : \mathsf{T}_\mathtt{c} = \mu\mathbf{t}.\ \&\mathtt{s}?\mathtt{incr}(\mathtt{int}).\, \textstyle\sum\big\{\ \underbrace{\oplus\mathtt{u}!\mathtt{res}\langle\mathtt{int}\rangle.\,\mathbf{t}}_{x < 2^8}\ ,\ \underbrace{\oplus\{\,0.8 : \mathtt{u}!\mathtt{res}\langle\mathtt{int}\rangle.\,\mathbf{t},\ \ 0.2 : \mathtt{s}!\mathtt{error}\langle\mathtt{int}\rangle.\,\mathbf{t}\,\}}_{\text{else, flip}(0.8)\,\{\mathrm{H}\Longrightarrow \ldots\ []\ \mathrm{T}\Longrightarrow \ldots\}}\ \big\}$$

Note that the condition in the *if*-statement of $P_\mathtt{c}$, depending on the size of the input x, is abstracted to a *non-deterministic choice* over the two output distributions — rendering *safety*, *deadlock-freedom* and *liveness*, undecidable at the process level (Prop. 2.3), decidable. Observe also that the choice of

$\mathbf{c}$ is *separated*: within one distribution, the res branch answers $\mathbf{u}$ while the error branch targets $\mathbf{s}$ — expressible because our selections fix a participant per branch, not per choice.

Probabilistic subtyping. Subtyping $\mathsf{T} \leqslant \mathsf{T}'$ allows a process typed by a smaller type T to stand in wherever one typed by T' is expected (the substitution principle [44]): it narrows selections (fewer nondeterministic options) and widens branchings (more offered inputs), as the rules make precise.

Definition 3.2 (Probabilistic subtyping). *Probabilistic subtyping* $\leqslant$ is the largest relation on session types (taken up to unfolding), coinductively defined by the rules below:

$$\text{[sub-}\oplus\text{]}\quad \frac{\forall i \in I:\ \mathsf{T}_i \leqslant \mathsf{T}'_i}{\oplus_{i\in I}\, p_i : \mathbf{p}_i!\ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i \leqslant \oplus_{i\in I}\, p_i : \mathbf{p}_i!\ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}'_i} \qquad \text{[sub-}\&\text{]}\quad \frac{\forall i \in I:\quad \mathsf{T}_i \leqslant \mathsf{T}'_i \qquad \{\mathbf{p}_i\}_{i\in I\cup J} = \{\mathbf{p}_i\}_{i\in I}}{\&_{i\in I\cup J}\, \mathbf{p}_i?\ell_i(\mathsf{B}_i).\mathsf{T}_i \leqslant \&_{i\in I}\, \mathbf{p}_i?\ell_i(\mathsf{B}_i).\mathsf{T}'_i}$$

$$\text{[sub-end]}\quad \frac{-}{\mathsf{end} \leqslant \mathsf{end}} \qquad \text{[sub-}\Sigma\text{]}\quad \frac{|I|\cdot|K| > 1 \quad \forall i \in I : \exists k \in K :\ \oplus_{j\in J_i}\, p_{i,j} : \mathbf{p}_j!\ell'_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j} \leqslant \oplus_{j\in J'_k}\, p_{k,j} : \mathbf{p}_j!\ell'_j\langle \mathsf{B}_j\rangle.\mathsf{T}'_{k,j}}{\sum_{i\in I}\oplus_{j\in J_i}\, p_{i,j} : \mathbf{p}_j!\ell'_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j} \leqslant \sum_{k\in K}\oplus_{j\in J'_k}\, p_{k,j} : \mathbf{p}_j!\ell'_j\langle \mathsf{B}_j\rangle.\mathsf{T}'_{k,j}}$$

The rules adapt the multiparty subtyping of [22] to fit our probabilistic types. [sub-end] relates end only to itself. [sub-&] is contravariant: the subtype may offer *more* inputs ($I \cup J$) than the supertype (I), as a process handling extra messages is safe where fewer are expected. As in the input subtyping rule [s-in] in [51, Def. 5], we require the set of participants $\{\mathbf{p}_i\}_{i\in I}$ be identical to $\{\mathbf{p}_i\}_{i\in I\cup J}$ for downward-closure for typing context properties ([36, Lemma C.2, Cor. C.18]). [sub-$\oplus$] relates two selection distributions only when targets, labels, sorts, and probabilities coincide (changing any would be observable and break substitutability). Rule [sub-Σ] is less restrictive in comparison: every summand of the subtype must embed (via [sub-$\oplus$]) into some summand of the supertype, so the supertype's non-deterministic options cover the subtype's; the premise $|I|\cdot|K| > 1$ prevents degenerate derivations (closing coinductive derivation after 1 step of [sub-Σ]).

Lemma 3.3 ($\leqslant$-preorder). *$\leqslant$ is reflexive and transitive.*

Example 3.4 (Subtyping). Consider the following subprocess of some larger session:

P_{cond} = if $x > 3$ then $\mathbf{p}!\mathsf{hi}\langle\mathsf{true}\rangle.\mathbf{0}$ else $\mathbf{q}!\mathsf{bye}\langle 1\rangle.\mathbf{0}$, where the types for the "then" and "else" branch are $\mathsf{T}_{\mathsf{then}} = \oplus\mathbf{p}!\mathsf{hi}\langle\mathsf{bool}\rangle.\,\mathsf{end}$ and $\mathsf{T}_{\mathsf{else}} = \oplus\mathbf{q}!\mathsf{bye}\langle\mathsf{int}\rangle.\,\mathsf{end}$. Then by [sub-$\Sigma$], we have $\mathsf{T}_{\mathsf{then}} \leqslant \sum\{\mathsf{T}_{\mathsf{then}}, \mathsf{T}_{\mathsf{else}}\}$ and $\mathsf{T}_{\mathsf{else}} \leqslant \sum\{\mathsf{T}_{\mathsf{then}}, \mathsf{T}_{\mathsf{else}}\}$. Thus, by [t-sub] and [t-cond] (both introduced later), we are able to type P_{cond} with the common supertype $\mathsf{T}_{cond} = \sum\{\mathsf{T}_{\mathsf{then}}, \mathsf{T}_{\mathsf{else}}\}$.

Proposition 3.5 ($\leqslant$-decidable). *Given T_1 and T_2, checking $\mathsf{T}_1 \leqslant \mathsf{T}_2$ is decidable.*

Definition 3.6 ($\leqslant$-least upper-bound). $\cdot \sqcup \cdot = \cdot$ is a partial relation on session types with $\mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}$ defined exactly when $\mathsf{T}_1 \leqslant \mathsf{T}$ and $\mathsf{T}_2 \leqslant \mathsf{T}$ and $\forall \mathsf{T}' :\ (\mathsf{T}_1 \leqslant \mathsf{T}' \wedge \mathsf{T}_2 \leqslant \mathsf{T}') \Rightarrow \mathsf{T} \leqslant \mathsf{T}'$

We write $\mathsf{T}_1 \sqcup \mathsf{T}_2$ for any T such that $\mathsf{T}_1 \sqcup \mathsf{T}_2 = \mathsf{T}$ and note that T is unique up to mutual subtyping.

Definition 3.7 (Unfolding). We define $\mathsf{unf}(\cdot)$ as $\mathsf{unf}(\mu \mathbf{t}.\mathsf{T}) = \mathsf{T}\{^{\mu \mathbf{t}.\mathsf{T}}/_{\mathbf{t}}\}$ and $\mathsf{unf}(\mathsf{T}) = \mathsf{T}$ otherwise.

Probabilistic merging. Subtyping allows us to type if-statements by widening the type sufficiently to accommodate both possible behaviours. Analogously, *merging* will allow us to reconcile the behaviour of both head and tail branches of a $\mathsf{flip}(p)\{\text{H}\Rightarrow \ldots \,\|\, \text{T}\Rightarrow \ldots\}$.

Definition 3.8 (Probabilistic merge). The merge is the largest relation on session types, such that whenever $\mathsf{T}_1 \sqcap_p \mathsf{T}_2 = \mathsf{T}$ the clauses below hold, with other cases being undefined:

[m-end] If $\mathsf{unf}(\mathsf{T}_1) = \mathsf{unf}(\mathsf{T}_2) = \mathsf{end}$ then $\mathsf{unf}(\mathsf{T}) = \mathsf{end}$

[M-&] If $\mathrm{unf}(\mathsf{T}_1) = \&_{i\in I}\, \mathtt{q_i}?\ell_i(\mathsf{B}_i).\mathsf{T}_i$ and $\mathrm{unf}(\mathsf{T}_2) = \&_{j\in J}\, \mathtt{q_j}?\ell_j(\mathsf{B}_j).\mathsf{T}'_j$ with $K = \{k \in I \cap J \mid \exists \mathsf{T} : \mathsf{T}_k \sqcup \mathsf{T}'_k = \mathsf{T}\}$ and $\{\mathtt{q_i}\}_{i\in I} = \{\mathtt{q_j}\}_{j\in J} = \{\mathtt{q_k}\}_{k\in K}$ then $\mathrm{unf}(\mathsf{T}) = \&_{k\in K}\, \mathtt{q_k}?\ell_k(\mathsf{B}_k).(\mathsf{T}_k \sqcup \mathsf{T}'_k)$

[M-⊕] If $\mathrm{unf}(T_1) = \oplus_{i\in I}\, p_i{:}\,\mathtt{q_i}!\ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i$ and $\mathrm{unf}(T_2) = \oplus_{j\in J}\, p'_j{:}\,\mathtt{q_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}'_j$ with $\forall k \in I \cap J : \exists \mathsf{T} : \mathsf{T}_k \sqcup \mathsf{T}'_k = \mathsf{T}$ then $\mathrm{unf}(T) = \oplus_{k\in I\cup J}\, p''_k{:}\,\mathtt{q_k}!\ell_k\langle \mathsf{B}_k\rangle.\mathsf{T}''_k$ and for every $k \in I \cup J$:

$$\begin{array}{rcl} k \in I \setminus J & \Longrightarrow & p''_k = p \cdot p_k \wedge \mathsf{T}''_k = \mathsf{T}_k \\ k \in J \setminus I & \Longrightarrow & p''_k = (1-p) \cdot p'_k \wedge \mathsf{T}''_k = \mathsf{T}'_k \\ k \in I \cap J & \Longrightarrow & p''_k = p \cdot p_k + (1-p) \cdot p'_k \wedge \mathsf{T}''_k = \mathsf{T}_k \sqcup \mathsf{T}'_k \end{array}$$

[M-Σ] If $\mathrm{unf}(\mathsf{T}_1) = \sum_{i\in I} \mathsf{T}_i$ and $\mathrm{unf}(\mathsf{T}_2) = \sum_{j\in J} \mathsf{T}'_j$ with $|I| \cdot |J| > 1$ then $\mathrm{unf}(\mathsf{T}) = \sum_{(i,j)\in I\times J} \mathsf{T}_{i,j}$ where $\mathsf{T}_i \sqcap_p \mathsf{T}'_j = \mathsf{T}_{i,j}$ for all $(i,j) \in I \times J$

The above merging operator is analogous to the binary session operator $\mathsf{T}_1 \boxplus_p \mathsf{T}_2$ of [34], but extended to multiparty session types. By [M-end] type end is mergeable only with end. Merging of external choice [M-&] must preserve the participant sets and drops any branches indexed by $\mathtt{p_i}?\ell_i$ when $i \notin I \cap J$ (analogous to the contravariance of [SUB-&]), or whose continuations T_i and T'_i have no common supertype — continuations are then typed by their least supertype $\mathsf{T}_k \sqcup \mathsf{T}'_k$. Rule [M-⊕] merges two selections, assigning to $\mathtt{p_i}!\ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i$ in $i \in I \setminus J$ (resp. $i \in J \setminus I$) probability $p \cdot p_i$ (resp. $(1-p) \cdot p'_i$), while the probability for $i \in I \cap J$ is given by the convex combination $p \cdot p_i + (1-p) \cdot p'_i$ with continuations typed by the least upper bound $\mathsf{T}_k \sqcup \mathsf{T}'_k$. Finally, the rule [M-Σ] merges every selection combination, the product $I \times J$ ensures every possible resolution of non-determinism is preserved; the side-condition $|I| \cdot |J| > 1$ prevents degenerate derivations.

Notation 3.9 (Multi-way merge). We define $\bigsqcap_{i\in I}(\mathsf{T}_i, p_i)$ inductively as: T_i if $p_i = 1$ and $|I| = 1$; $\bigsqcap_{i\in I}(\mathsf{T}_i, p_i) = \mathsf{T}_j \sqcap_{p_j} \bigsqcap_{i\in I\setminus\{j\}}(\mathsf{T}_i, \frac{p_i}{1-p_j})$ if $j \in I \wedge p_j \neq 0$; undefined otherwise.

Example 3.10 (Probabilistic merging). The non-deterministic choice in the type $\mathsf{T}_\mathtt{c}$ of the calculator (Ex. 3.1) contains the selection distribution $\mathsf{T}_{\mathtt{c}-\mathsf{else}} = \oplus\{\, 0.8 : \mathtt{u}!\mathsf{res}\langle \mathtt{int}\rangle.\,\mathbf{t},\ \ 0.2 : \mathtt{s}!\mathsf{error}\langle \mathtt{int}\rangle.\,\mathbf{t}\,\}$, which is the merge $\mathsf{T}_1 \sqcap_{0.8} \mathsf{T}_2$ of $\mathsf{T}_1 = \oplus 1 : \mathtt{u}!\mathsf{res}\langle \mathtt{int}\rangle.\,\mathbf{t}$ and $\mathsf{T}_2 = \oplus 1 : \mathtt{s}!\mathsf{error}\langle \mathtt{int}\rangle.\,\mathbf{t}$ (with $\mathbf{t}$ of type $\mathsf{T}_\mathtt{c}$). Since the label-role pairs $\mathtt{u}!\mathsf{res}$ and $\mathtt{s}!\mathsf{error}$ are disjoint, a single application of [M-⊕] suffices (as $0.8 = 0.8 \cdot 1$, $0.2 = 0.2 \cdot 1$ and $\mathbf{t} = \mathbf{t}$, by the disjoint case of the statement).

Proposition 3.11 (Decidability of merge). *For $\mathsf{T}_1, \mathsf{T}_2$ and $p \in (0,1)$ it is decidable whether $\mathsf{T}_1 \sqcap_p \mathsf{T}_2$ is defined in time $O(|\mathsf{T}_1||\mathsf{T}_2|)$, and when defined the witness is computable in time $O\big((\max\{2,d\})^{2|\mathsf{T}_1||\mathsf{T}_2|} \log(|\mathsf{T}_1||\mathsf{T}_2|+1)\big)$ where d is the max out-degree of the type graphs of $\mathsf{T}_1, \mathsf{T}_2$.*

A typing context $(\Delta, \Delta', \dots)$ is a mapping from a participant to its type, representing behaviours of a multiparty session M. Its behaviour (semantics) are defined by a labelled transition relation labelled with *actions* $(\alpha, \alpha', \dots)$. Typing contexts and actions are defined as:

$$\Delta ::= \mathtt{p} : \mathsf{T} \mid \Delta, \Delta' \qquad \alpha ::= \mathtt{p}{:}\mathtt{q}!\ell\langle \mathsf{B}\rangle \mid \mathtt{p}{:}\mathtt{q}?\ell(\mathsf{B}) \mid (\mathtt{p},\mathtt{q})\ell \mid \mathtt{p}{:}\tau \mid \mathtt{p}{:}p$$

Action $\mathtt{p}{:}\mathtt{q}!\ell\langle \mathsf{B}\rangle$ is a *selection* from participant $\mathtt{p}$ to $\mathtt{q}$ with label ℓ and type B; action $\mathtt{p}{:}\mathtt{q}?\ell(\mathsf{B})$ is *branching* at participant $\mathtt{p}$ from $\mathtt{q}$ with label ℓ and type B; and action $(\mathtt{p},\mathtt{q})\ell$ is a *communication* between dual actions. $\mathtt{p}{:}\tau$ and $\mathtt{p}{:}p$ denote *nondeterministic* and *probabilistic* internal choice respectively. We write $\Delta_1 \leqslant \Delta_2$ if $\mathrm{dom}(\Delta_1) = \mathrm{dom}(\Delta_2)$ and $\forall \mathtt{p} \in \mathrm{dom}(\Delta_1) : \Delta_1(\mathtt{p}) \leqslant \Delta_2(\mathtt{p})$.

Semantics (Labelled Transition System) of typing contexts is defined in Fig. 4. We note that emitting a message decomposes into up to *three* steps: [CT-ND] resolves an internal *non-deterministic* choice among the summands of a Σ; [CT-PROB] is an internal *probabilistic* reduction, resolving a participant $\mathtt{p}$ at a *proper* output selection $\mathtt{p} : \oplus_{i\in I}\, p_i{:}\,\mathtt{q_i}!\ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i$ to the singleton selection of branch k, with probability p_k; this enables [CT-OUT], emitting the message to $\mathtt{q_k}$ with label ℓ_k, sort B_k, and probability 1. Rule [CT-IN] inputs label ℓ_k with sort B_k at participant $\mathtt{p}$ with probability 1; rule [CT-τ]

$$[\text{CT-REC}]\ \frac{\mathtt{p} : \mathsf{T}\{\mu \mathtt{t}.\mathsf{T}/\mathtt{t}\} \xrightarrow{\alpha}_t \mathtt{p} : \mathsf{T}' \quad t \in (0;1] \cup \{n\}}{\mathtt{p} : \mu \mathtt{t}.\mathsf{T} \xrightarrow{\alpha}_t \mathtt{p} : \mathsf{T}'} \qquad [\text{CT-ND}]\ \frac{k \in I \quad |I| > 1}{\mathtt{p} : \textstyle\sum_{i\in I} \bigoplus_{j\in J_i} p_{i,j} : \mathtt{q_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{i,j} \xrightarrow{\mathtt{p}:\tau}_n \mathtt{p} : \bigoplus_{j\in J_k} p_{k,j} : \mathtt{q_j}!\ell_j\langle \mathsf{B}_j\rangle.\mathsf{T}_{k,j}}$$

$$[\text{CT-PROB}]\ \frac{k \in I \qquad p_k \in (0;1)}{\mathtt{p} : \bigoplus_{i\in I} p_i : \mathtt{q_i}!\ell_i\langle \mathsf{B}_i\rangle.\mathsf{T}_i \xrightarrow{\mathtt{p}:p}_{p_k} \mathtt{p} : \bigoplus 1 : \mathtt{q_k}!\ell_k\langle \mathsf{B}_k\rangle.\mathsf{T}_k} \qquad [\text{CT-OUT}]\ \frac{}{\mathtt{p} : \bigoplus 1 : \mathtt{q}!\ell\langle \mathsf{B}\rangle.\mathsf{T} \xrightarrow{\mathtt{p}:\mathtt{q}!\ell\langle \mathsf{B}\rangle}_1 \mathtt{p} : \mathsf{T}}$$

$$[\text{CT-IN}]\ \frac{k \in I}{\mathtt{p} : \&_{i\in I}\ \mathtt{q_i}?\ell_i(\mathsf{B}_i).\mathsf{T}_i \xrightarrow{\mathtt{p}:\mathtt{q_k}?\ell_k(\mathsf{B}_k)}_1 \mathtt{p} : \mathsf{T}_k} \qquad [\text{CT-}\tau]\ \frac{\Delta_1 \xrightarrow{\mathtt{p}:\mathtt{q}!\ell\langle \mathsf{B}\rangle}_1 \Delta_1' \quad \Delta_2 \xrightarrow{\mathtt{q}:\mathtt{p}?\ell(\mathsf{B})}_1 \Delta_2'}{\Delta_1, \Delta_2 \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1 \Delta_1', \Delta_2'} \qquad [\text{CT-SC}]\ \frac{\Delta \xrightarrow{\alpha}_t \Delta' \quad t \in (0;1] \cup \{n\}}{\Delta, \mathtt{p} : \mathsf{T} \xrightarrow{\alpha}_t \Delta', \mathtt{p} : \mathsf{T}}$$

Fig. 4. Labelled transition system for typing contexts

synchronises a singleton selection with its dual branching; [CT-REC] is the standard rule for recursive types; and [CT-SC] is a context rule. We write:

- $\Delta \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_p \Delta'$ iff $(p = 1 \wedge \Delta \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1 \Delta')$ or $\exists \Delta'' : \Delta \xrightarrow{\mathtt{p}:p}_p \Delta'' \wedge \Delta'' \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1 \Delta'$.
- $\Delta \to_t \Delta'$ iff $\exists \alpha \in \{\mathtt{p}{:}\tau, \mathtt{p}{:}p, (\mathtt{p},\mathtt{q})\ell \mid \mathtt{p},\mathtt{q},\ell\} : \Delta \xrightarrow{\alpha}_t \Delta'$; and $\Delta \to \Delta'$ iff $\exists t \in (0;1] \cup \{n\} : \Delta \to_t \Delta'$; and $\Delta \xrightarrow{\alpha}_t$ iff $\exists \Delta' : \Delta \xrightarrow{\alpha}_t \Delta'$ (same for $\to_t$ and $\to$). These conventions also apply to $\xrightarrow{\alpha}_p$.
- $\Delta_1 \longrightarrow_p^+ \Delta_m$ for the *transitive closure*: a sequence $\Delta_1 \cdots \longrightarrow_{p_{m-1}} \Delta_m$ with $m \geq 2$ and $p = \prod_{k=1}^{m-1} p_k$. The *reflexive* and *transitive closure* $\longrightarrow_p^*$ additionally admits the empty path ($\Delta \longrightarrow_1^* \Delta$).

Safety property. In a general typing system [52], we type-check that a multiparty session M satisfies a *safety property*: whether its typing context only exchanges *compatible* labels.

Definition 3.12 (Safety). [52, Definition 4.1] φ is a *safety property* of Δ iff, whenever $\varphi(\Delta)$: (1) $\varphi(\emptyset)$; (2) $\Delta \xrightarrow{\mathtt{p}:\mathtt{q}!\ell\langle \mathsf{B}\rangle}_1$ and $\Delta \xrightarrow{\mathtt{q}:\mathtt{p}?\ell'(\mathsf{B}')}_1$ imply $\Delta \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1$; (3) $\Delta \to \Delta'$ implies $\varphi(\Delta')$ We write safe(Δ) if $\varphi(\Delta)$ holds for some safety property φ.

Def. 3.12(1) above states the empty context is considered safe, (2) defines that if Δ contains the entries for selection at $\mathtt{p}$ to $\mathtt{q}$ and branching at $\mathtt{q}$ from $\mathtt{p}$, then branching at $\mathtt{q}$ supports *all* of $\mathtt{p}$'s labels. Def. 3.12(3) states that a safety property must be preserved under reductions.

Example 3.13 (Semantics of types and safety). Consider the typing context $\Delta = \mathtt{u} : \mathsf{T}_\mathtt{u}, \mathtt{s} : \mathsf{T}_\mathtt{s}, \mathtt{c} : \mathsf{T}_\mathtt{c}$ of Example 3.1. We consider two runs starting from Δ:

$$\Delta \xrightarrow{(\mathtt{u},\mathtt{s})\text{incr}}_1 \cdot \xrightarrow{(\mathtt{s},\mathtt{c})\text{incr}}_1 \cdot \xrightarrow{\mathtt{c}:\tau}_n \cdot \xrightarrow{(\mathtt{c},\mathtt{s})\text{error}}_{0.2} \cdot \xrightarrow{(\mathtt{s},\mathtt{c})\text{incr}}_1 \cdot \xrightarrow{\mathtt{c}:\tau}_n \cdot \xrightarrow{(\mathtt{c},\mathtt{u})\text{res}}_1 \Delta \cdots$$

$$\Delta \xrightarrow{(\mathtt{u},\mathtt{s})\text{incr}}_1 \cdot \xrightarrow{(\mathtt{s},\mathtt{c})\text{incr}}_1 \cdot \xrightarrow{\mathtt{c}:\tau}_n \cdot \xrightarrow{(\mathtt{c},\mathtt{s})\text{error}}_{0.2} \cdot \xrightarrow{(\mathtt{s},\mathtt{c})\text{incr}}_1 \cdot \xrightarrow{\mathtt{c}:\tau}_n \cdot \xrightarrow{(\mathtt{c},\mathtt{s})\text{error}}_{0.2} \cdot \xrightarrow{(\mathtt{s},\mathtt{c})\text{incr}}_1 \cdot \xrightarrow{\mathtt{c}:\tau}_n \cdot \xrightarrow{(\mathtt{c},\mathtt{s})\text{error}}_{0.2} \cdots$$

The first run fails once then succeeds when the [CT-ND] step commits $\mathtt{c}$'s non-deterministic choice to the selection sending res to $\mathtt{u}$ with probability 1. The second fails indefinitely, a run of probability $\lim_{n\to\infty} 0.2^n = 0$ (§ 4). Note the $\xrightarrow{\mathtt{c}:\tau}_n$ steps carry no probability, and instead abstract over the $x < 2^8$ test in Example 3.1.

3.2 Probabilistic Multiparty Session Typing System

We introduce the probabilistic multiparty session typing system. We define *typing environments* ($\Gamma, \Gamma', \ldots$) as a mapping from a variable to a sort for typing expressions and a process variable to a

$$[\text{T-REC}]\ \frac{\Gamma, X : \mathsf{T} \vdash P : \mathsf{T}}{\Gamma \vdash \mu X.P : \mathsf{T}} \qquad [\text{T-VAR}]\ \frac{}{\Gamma, X : \mathsf{T} \vdash X : \mathsf{T}} \qquad [\text{T-SUB}]\ \frac{\Gamma \vdash P : \mathsf{T} \quad \mathsf{T} \leqslant \mathsf{T}'}{\Gamma \vdash P : \mathsf{T}'}$$

$$[\text{T-0}]\ \frac{}{\Gamma \vdash \mathbf{0} : \mathsf{end}} \qquad [\text{T-FLIP}]\ \frac{\Gamma \vdash P_i : \mathsf{T}_i \quad i \in \{1,2\}}{\Gamma \vdash \mathsf{flip}(p)\{\text{H}\Rightarrow P_1 \parallel \text{T}\Rightarrow P_2\} : \mathsf{T}_1 \sqcap_p \mathsf{T}_2}$$

$$[\text{T-SEL}]\ \frac{\Gamma \vdash \mathsf{e} : \mathsf{B} \quad \Gamma \vdash P : \mathsf{T}}{\Gamma \vdash \mathsf{p}!\ell\langle \mathsf{e}\rangle.P : \oplus\, 1 : \mathsf{p}!\ell\langle \mathsf{B}\rangle.\mathsf{T}} \qquad [\text{T-BRA}]\ \frac{\forall i \in I \quad \Gamma, x_i : \mathsf{B}_i \vdash P_i : \mathsf{T}_i}{\Gamma \vdash \Sigma_{i\in I}\mathsf{p}_i?\ell_i(x_i).P_i : \&_{i\in I}\ \mathsf{p}_i?\ell_i(\mathsf{B}_i).\mathsf{T}_i}$$

$$[\text{T-COND}]\ \frac{\Gamma \vdash \mathsf{e} : \mathsf{bool} \quad \Gamma \vdash P_1 : \mathsf{T} \quad \Gamma \vdash P_2 : \mathsf{T}}{\Gamma \vdash \mathsf{if}\ \mathsf{e}\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : \mathsf{T}} \qquad [\text{B-SESS}]\ \frac{\mathsf{safe}(\{\mathsf{p}_i : \mathsf{T}_i\}_{i\in I}) \quad \forall i \in I : \vdash P_i : \mathsf{T}_i}{\vdash \prod_{i\in I}\mathsf{p}_i \triangleleft P_i : \{\mathsf{p}_i : \mathsf{T}_i\}_{i\in I}}$$

Fig. 5. Probabilistic multiparty session typing system

session type for typing recursive processes: $\Gamma ::= \varnothing \mid \Gamma, x : \mathsf{B} \mid \Gamma, X : \mathsf{T}$. The judgements for typing expressions, processes and sessions are written as $\Gamma \vdash \mathsf{e} : \mathsf{B}$, $\Gamma \vdash P : \mathsf{T}$, and $\vdash M : \Delta$, respectively.

We omit the typing rules for expressions as they are standard and identical with the rules in [22]. Fig. 5 defines the typing rules for processes and a session. With the exception of [T-FLIP], [T-SEL] and [B-SESS], the rules are standard and identical with the corresponding rules in [22]. Rule [T-FLIP] is the main rule for assigning probabilities with respect to p to types using the probabilistic merging operator in Def. 3.8. Rule [T-SEL] types a single selection with probability 1. Rule [B-SESS] is identical with the bottom-up rule [B-SESS] in [59, Figure 5] where session M is typed by a safe typing context (note: [22] uses the top-down typing system).

Theorem 3.14 (Decidability). *Given session M and typing context Δ, $\vdash M : \Delta$ is decidable.*

Proof. The proof is similar to [52, Thm. 4.11] with $\leqslant$, $\sqcap_p$, and $\mathsf{safe}(\Delta)$ decidable by Props. 3.5 and 3.11 and [36, Prop. B.2]). (Full proof in [36, App. B.3]). □

4 Properties of Probabilistic Multiparty Sessions

This section studies a range of probabilistic properties of probabilistic multiparty session types and processes, after proving *subject reduction* and *session fidelity*. See [36, App. C] for proofs.

4.1 Subject Reduction and Session Fidelity Theorems

Without probabilities, subject reduction states that each process reduction is matched by a context transition. Flips complicate this: contexts have no flip transitions, since $\sqcap_p$ compiled each flip into new distributions at typing time, so a single probabilistic step cannot preserve typing alone. Instead, we look at the weighted *pair* of outcomes, recombined through the merge (case (3) below).

Theorem 4.1 (Subject Reduction). *Suppose $\vdash M : \Delta$. If $M \longrightarrow_p M_1$, then depending on the top-level rule used to derive $M \longrightarrow_p M_1$ we have:*

(1) [R-T-IF] *or* [R-F-IF] *implies:* $\vdash M_1 : \Delta$

(2) [R-COMM] *implies:* $\exists \Delta_1$ *such that* $\vdash M_1 : \Delta_1$ *and either* $\Delta \rightarrow_1 \Delta_1$ *or* $\Delta \rightarrow_n \cdot \rightarrow_1 \Delta_1$.

(3) [R-HEAD] *or* [R-TAIL] *implies* $\exists \mathsf{p}, \mathsf{T}, \{\mathsf{T}_i, M_i\}_{i\in\{1,2\}}, \Delta'$ *such that* $\Delta = \mathsf{p} : \mathsf{T}, \Delta'$ *and* $M \rightarrow_{1-p} M_2$ *with* $\vdash M_i : (\mathsf{p} : \mathsf{T}_i, \Delta')$ *for* $i \in \{1,2\}$ *and* $(\mathsf{p} : \mathsf{T}_1 \sqcap_p \mathsf{T}_2, \Delta') \leqslant \Delta$.

Proof. By induction on the derivation of $M \rightarrow_p M_1$; see [36, App. C.2]. □

Case (1) covers data-dependent control-flow steps. In case (2) the context matches the communication, possibly after first committing its nondeterministic choice via $\rightarrow_n$. In case (3), as no context transition matches a flip, the statement instead mentions *both* outcomes, whose residual typings recombine into $\mathsf{T}_1 \sqcap_p \mathsf{T}_2 \leqslant \mathsf{T}$. Case (3) is illustrated in Fig. 6: merging the typings at each frontier of

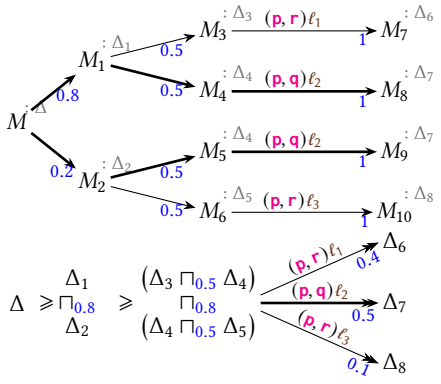


Fig. 6. Subject reduction: The session M reduces probabilistically to one of $M_3, \ldots, M_6$ before $\mathtt{p}$ sends a message to $\mathtt{q}$ or $\mathtt{r}$. Both M_4, M_5 (and likewise M_8, M_9) are typed by the same context. Thm. 4.1(3) states that merging the typings $\mathsf{T}_1, \mathsf{T}_2$ of a participant $\mathtt{p}$ across both outcomes of its flip always produces a subtype of its type in Δ ($\mathsf{T}_1 \sqcap_p \mathsf{T}_2 \leqslant \mathsf{T}$). Under the merged context (bottom), $\mathtt{p}$ holds a distribution over its sends.

Session fidelity (Thm. 4.4(3)) recovers the resolving step's probability as the total weight of the flip paths exposing that send: for $(\mathtt{p},\mathtt{q})\ell_2$, $0.8 \cdot 0.5 + 0.2 \cdot 0.5 = 0.5$ (bold edges). The resolved context mirrors Δ_4, which types both M_4 and M_5.

a flip tree yields a context that is a subtype of Δ (the composed chain follows by monotonicity of merging Lemma 4.20).

Corollary 4.2 (Communication Safety). *If* $\vdash M : \Delta$, *then* $M \not\twoheadrightarrow^*$ error.

Session fidelity states the converse: typing context reductions must be matched by the process.

Notation 4.3 (Annotated session reductions). We write $M \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1 M'$ when $M \longrightarrow_1 M'$ is derived by top-level rule [r-comm] exchanging label ℓ from $\mathtt{p}$ to $\mathtt{q}$.

Theorem 4.4 (Session Fidelity).

(1) If $\vdash M : \Delta$ *then* $\Delta \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1 \Delta'$ *implies* $\exists I : \{M \longrightarrow^*_{p_i} M_i \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1\}_{i\in I}$ *with* $\sum_{i\in I} p_i = 1$

(2) If $\vdash P : \mathsf{T}$ *then* $\mathtt{p} : \mathsf{T} \xrightarrow{\mathtt{p}:\tau}_n$ *implies* $\exists \mathsf{T}' : \mathtt{p} : \mathsf{T} \xrightarrow{\mathtt{p}:\tau}_n \mathtt{p} : \mathsf{T}'$ *and* $\exists I : \{\mathtt{p} \triangleleft P \longrightarrow^*_{p_i} \mathtt{p} \triangleleft P_i \mid \mathtt{p} \triangleleft P_i \not\twoheadrightarrow\}_{i\in I}$ *with* $\sum_{i\in I} p_i = 1$ *and a set of types* $\{\mathsf{T}_i \mid \vdash P_i : \mathsf{T}_i\}_{i\in I}$ *such that* $\bigsqcap_{i\in I}(\mathsf{T}_i, p_i) \leqslant \mathsf{T}'$

(3) If $\vdash P : \mathsf{T}$ *then* $\mathtt{p} : \mathsf{T} \xrightarrow{\mathtt{p}:p}_p \mathtt{p} : \bigoplus 1{:}\,\mathtt{q}!\ell\langle \mathsf{B}\rangle.\mathsf{T}'$ *implies* $\exists I : \{\mathtt{p} \triangleleft P \longrightarrow^*_{p_i} \mathtt{p} \triangleleft \mathtt{q}!\ell\langle \mathtt{e}_i\rangle.P_i\}_{i\in I}$ *with* $\sum_{i\in I} p_i = p$ *and* $\vdash P_i : \mathsf{T}'$ *for all* $i \in I$

(2) and (3) are easily lifted to $\vdash M : \Delta$ through [ct-sc].

Proof. Corollary of a strengthened statement, proved by induction; see [36, App. C.2]. □

Case (1) states that a context communication of probability 1 is realised by weighted paths $M \longrightarrow^*_{p_i} M_i$, each arriving where the communication fires surely, with $\sum_{i\in I} p_i = 1$. Case (2) states that when the context commits its nondeterministic choice, the process resolves its flips down to leaves $\{P_i\}_{i\in I}$ that reduce no further, and merging the leaf types at their path weights, $\bigsqcap_{i\in I}(\mathsf{T}_i, p_i)$, recovers a subtype of the committed type. Analogously, case (3) guarantees that $\mathtt{p}$'s probabilistic transition to $\mathtt{p} : \bigoplus 1{:}\,\mathtt{q}!\ell\langle \mathsf{B}\rangle.\mathsf{T}'$ in the typing context corresponds to the probability of reducing to a leaf in $\{\mathtt{p} \triangleleft \mathtt{q}!\ell\langle \mathtt{e}_i\rangle.P_i\}_{i\in I}$, and that the continuation P_i is typed by T'. This is illustrated in Fig. 6, with the bold paths representing $(\mathtt{p},\mathtt{q})\ell_2$ with weights $0.4 + 0.1 = 0.5$.

4.2 Paths and Probabilities

To take probabilities into account, we formalise the notion of *paths* and the probability of obtaining a set of paths. Since typing contexts combine probabilistic choice with nondeterminism, such probabilities are well-defined only relative to a *scheduler* that resolves the nondeterminism. We therefore interpret a typing context as a finite Markov Decision Process (MDP), define the probability

of a set of paths under a fixed scheduler by standard means [37], and obtain lower bounds that transfer to well-typed sessions by taking the *minimum* over all (fair) schedulers §4.4.

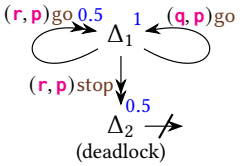


Fig. 7. A deadlock dependent on scheduling.

A typing context can be seen as inducing a finite MDP — an LTS with *nondeterministic* and *probabilistic* state transitions. We distinguish between *two* sources of nondeterminism: (1); internal Σ resolutions $\mathsf{p}{:}\sum_{i\in I}\mathsf{T}_i \xrightarrow{\mathsf{p}:\tau}_n \mathsf{p}{:}\mathsf{T}_i$, and (2); scheduling of parallel enabled roles — as *separated choice* means that a participant p may be simultaneously waiting to receive from both q and r, and both being enabled, the scheduler must choose which pair gets to communicate, consider for instance Fig. 7 — if r is scheduled to send there is a 0.5 chance of *deadlocking* (r sends stop to p and both quit, leaving q hanging), scheduling q on the other hand sends go to p and recurses. Thus scheduling of concurrently enabled participants may affect the probability of satisfying a property. Later, we will work with a restricted class of *fair* schedulers that cannot starve persistently enabled actions.

Definition 4.5 (Path). A path ξ is a (potentially infinite) sequence of contexts $\Delta_1;\Delta_2;\cdots$ such that for all $i < |\xi|$ we have $\Delta_i \to_p \Delta_{i+1}$ or $\Delta_i \to_n \Delta_{i+1}$. A path ξ is *complete* if it is either infinite or $\Delta_{|\xi|} \not\to$. We write $\Xi(\Delta)$ for the set of all complete paths starting from Δ.

Definition 4.6 (Scheduler). A scheduler $\phi \in \Phi^\Delta$ for typing context Δ is a function mapping each finite path $\Delta_1;\cdots;\Delta_n$ of Δ with $\Delta_n \to$ to a pair (p, Δ), of a participant and an optional typing context, such that: (1) if $\phi(\Delta_1;\cdots;\Delta_n) = (\mathsf{p}, \emptyset)$, then either $\Delta_n \xrightarrow{\mathsf{p}:p}_p$ for some p, or $\Delta_n \xrightarrow{(\mathsf{p},\mathsf{q})\ell}_1$ for some q and ℓ; and (2) if $\phi(\Delta_1;\cdots;\Delta_n) = (\mathsf{p}, \Delta_{n+1})$, then $\Delta_n \xrightarrow{\mathsf{p}:\tau}_n \Delta_{n+1}$.

Fixing a scheduler $\phi \in \Phi^\Delta$ resolves all nondeterminism in the MDP of Δ, inducing a Discrete-time Markov chain (DTMC) in which every transition carries a probability (the scheduler induced transitions can be seen to carry probability 1). The probability of a *finite* path is then the product of its transition probabilities; we write $\Xi_\phi(\Delta)$ for the set of complete paths induced by ϕ from Δ. To lift finite-path probabilities to a measure on complete paths, we follow [37] and take cylinder sets as our basic events.

Definition 4.7 (Cylinder set). The *cylinder set* $\mathsf{Cyl}(\xi)$ of a finite path $\xi = \Delta_1;\cdots;\Delta_n$ of the DTMC induced by a scheduler ϕ is the set of all complete paths in $\Xi_\phi(\Delta_1)$ with prefix ξ; in particular, if ξ is itself complete, then $\mathsf{Cyl}(\xi) = \{\xi\}$. We write $\mathbb{P}^\phi_{\Delta_1}$ for the unique probability measure on the σ-algebra generated by the cylinder sets with $\mathbb{P}^\phi_{\Delta_1}(\mathsf{Cyl}(\xi)) = \prod_{i=1}^{n-1} p_i$, where each $\Delta_i \to_{p_i} \Delta_{i+1}$; measurable sets of paths are *events*.

For a predicate f on paths, we write $\mathbb{P}^\phi_\Delta(f) = \mathbb{P}^\phi_\Delta(\{\xi \in \Xi_\phi(\Delta) \mid f(\xi)\})$ for the probability of obtaining a path satisfying f under ϕ from Δ. For state predicate g we define $\Diamond\, g$ to be the event "eventually g", where $(\Diamond\, g)(\xi) \iff \exists i \le |\xi| : g(\xi[i])$.

4.3 Probabilistic Deadlock-Freedom and Liveness for Typing Contexts

In this section we define *two* properties of typing contexts, *deadlock-freedom* and the slightly stronger property of *liveness*. We first define the non-probabilistic version of each before introducing their probabilistic counterparts. A context Δ is *deadlocked* if it has no transitions and some participant has a pending action. A context is *deadlock-free* if no *deadlocked* state is reachable.

Definition 4.8 (Deadlock-freedom).

(1) A context Δ is *deadlocked*, written $\mathsf{deadlock}(\Delta)$, if $\Delta \not\to$ and $\exists \mathbf{p} : \Delta(\mathbf{p}) \neq \mathsf{end}$.
(2) A context Δ is *deadlock-free*, written $\mathsf{df}(\Delta)$, if $\Delta \to^* \Delta'$ implies $\neg\mathsf{deadlock}(\Delta')$.

Before introducing the concept of *liveness*, we define what it means for a path to be *fair*, as our notion of liveness depends upon the scheduler being fair.

Definition 4.9 (Fair path). A complete path ξ is *fair*, written $\mathsf{fair}(\xi)$, if the following hold:

(1) $\forall \mathbf{p}, \mathbf{q}, \ell : \text{if } \forall i\, \exists j \geq i : \Delta_j \xrightarrow{(\mathbf{p},\mathbf{q})\ell}_1 \text{ then } \forall i\, \exists n \geq i : \Delta_n \xrightarrow{(\mathbf{p},\mathbf{q})\ell}_1 \Delta_{n+1}$
(2) $\forall \mathbf{p}\, \forall i : \text{if } \Delta_i \xrightarrow{\mathbf{p}:\tau}_n \text{ then } \exists n \geq i : \Delta_n \xrightarrow{\mathbf{p}:\tau}_n \Delta_{n+1}$
(3) $\forall \mathbf{p}\, \forall i : \text{if } \Delta_i \xrightarrow{\mathbf{p}:p} \text{ then } \exists n \geq i, p : \Delta_n \xrightarrow{\mathbf{p}:p}_p \Delta_{n+1}$

Intuitively, a path is *fair* if a participant $\mathbf{p}$ cannot be denied sending an *enabled* communication action indefinitely; and if $\mathbf{p}$ has an internal step to take then it will eventually do so.

A path is then *live* if any $\mathbf{p}$ with a pending *input* or *output* action along it eventually communicates — note that this does not require the communication partner to be immediately available.

Definition 4.10 (Live path). A path ξ is *live*, written $\mathsf{live}(\xi)$, if for every $\Delta_i \in \xi$:

(1) if $\Delta_i \xrightarrow{\mathbf{p}:\mathbf{q}!\ell\langle \mathsf{B}\rangle}_1$, then $\exists n \geq i : \Delta_n \xrightarrow{(\mathbf{p},\mathbf{q})\ell}_1 \Delta_{n+1}$
(2) if $\Delta_i \xrightarrow{\mathbf{q}:\mathbf{p}?\ell(\mathsf{B})}_1$, then $\exists n \geq i, \mathbf{r}, \ell' : \Delta_n \xrightarrow{(\mathbf{r},\mathbf{q})\ell'}_1 \Delta_{n+1}$

Definition 4.11 (Live contexts). Given a context Δ, Δ is *live*, written $\mathsf{live}(\Delta)$, if every complete path $\xi \in \Xi(\Delta)$ that is fair is also live.

Example 4.12 (Non-liveness of the calculator service). Consider again the calculator service of Ex. 2.1, whose types are given in Ex. 3.1. The typing context formed by $\Delta = \mathbf{u}{:}\mathsf{T}_\mathbf{u}, \mathbf{s}{:}\mathsf{T}_\mathbf{s}, \mathbf{c}{:}\mathsf{T}_\mathbf{c}$ is *not live* according to Def. 4.11. To see this, note that there is an *infinite* reduction path

$$\Delta \xrightarrow{(\mathbf{u},\mathbf{s})\mathsf{incr}}_1 \cdot \xrightarrow{(\mathbf{s},\mathbf{c})\mathsf{incr}}_1 \cdot \xrightarrow{\mathbf{c}:\tau}_n \cdot \xrightarrow{(\mathbf{c},\mathbf{s})\mathsf{error}}_{0.2} \cdot \xrightarrow{(\mathbf{s},\mathbf{c})\mathsf{incr}}_1 \cdot \xrightarrow{\mathbf{c}:\tau}_n \cdot \xrightarrow{(\mathbf{c},\mathbf{s})\mathsf{error}}_{0.2} \cdot \xrightarrow{(\mathbf{s},\mathbf{c})\mathsf{incr}}_1 \cdot \xrightarrow{\mathbf{c}:\tau}_n \cdot \xrightarrow{(\mathbf{c},\mathbf{s})\mathsf{error}}_{0.2} \cdots$$

where after $\mathbf{u}$'s initial message, $\mathbf{u}$ is waiting at $\mathbf{u} \xrightarrow{\mathbf{u}:\mathbf{c}?\mathsf{res}(\mathsf{int})}_1$ while $\mathbf{s}$ and $\mathbf{c}$ are communicating indefinitely, yet the probability of seeing this path is $\lim_{n\to\infty} 0.2^n = 0$.

Ex. 4.12 illustrates that for *probabilistic* programs, it would be desirable to have more permissive notions of the properties, admitting *non-live* (or deadlocking) paths so long as they have measure 0 — or indeed any quantifiable measure. We therefore measure both properties, asking: what probability of deadlock-freedom, and of liveness, can be *guaranteed* however the nondeterminism is resolved? Since the probability of an event is well-defined only relative to a scheduler (Defs. 4.6 and 4.7), we quantify over those schedulers that induce *fair* paths with probability 1;

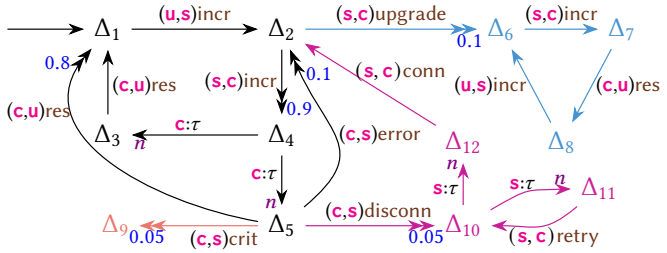
Definition 4.13 (Fair schedulers). A scheduler ϕ over Δ is *fair* if $\phi \in \Phi^\Delta_F = \{\phi \in \Phi^\Delta \mid \mathbb{P}^\phi_\Delta(\mathsf{fair}) = 1\}$

Definition 4.14 (Probabilistic deadlock-freedom and liveness). For a context Δ, we define the *minimum probability* of deadlock-freedom and of liveness as

$$\mathbb{P}^{\min}_{\mathsf{df}}(\Delta) = 1 - \max_{\phi \in \Phi^\Delta_F} \mathbb{P}^\phi_\Delta(\Diamond\, \mathsf{deadlock}) \qquad \mathbb{P}^{\min}_{\mathsf{live}}(\Delta) = \min_{\phi \in \Phi^\Delta_F} \mathbb{P}^\phi_\Delta(\mathsf{live})$$

Definition 4.15 (Almost-sure properties). Δ is *almost-surely deadlock-free*, written $\mathsf{asdf}(\Delta)$, if $\mathbb{P}^{\min}_{\mathsf{df}}(\Delta) = 1$, and *almost-surely live*, written $\mathsf{aslive}(\Delta)$, if $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta) = 1$.

Example 4.16 (Calculator service extended). Consider a refinement of the *calculator service* from Ex. 3.1 whose labelled transition system is given in Fig. 8. Δ_{1-5} correspond to the original session from Ex. 3.1, but the 0.2 probability of an error occurring has been distributed into *two*



	$\mathbb{P}^{\min}_{\text{live}}$	$\mathbb{P}^{\min}_{\text{df}}$		$\mathbb{P}^{\min}_{\text{live}}$	$\mathbb{P}^{\min}_{\text{df}}$
Δ_1	0.52	0.69	Δ_7	1	1
Δ_2	0.52	0.69	Δ_8	1	1
Δ_3	0.52	0.69	Δ_9	0	0
Δ_4	0.47	0.65	Δ_{10}	0	0.69
Δ_5	0.47	0.65	Δ_{11}	0	0.69
Δ_6	1	1	Δ_{12}	0.52	0.69

Fig. 8. A refined calculator service Ex. 4.16.

additional events now, (1); with probability 0.05 the error is *critical* and c informs s after which both quit, leaving u hanging (a deadlock, Δ_9), (2); with probability 0.05, c loses its connection (disconn) to s, entering a retry loop (Δ_{10-11}), eventually it *may* reconnect (Δ_{12}) but this is *not* guaranteed. Finally, s knows c is faulty, and will with probability 0.1 upgrade to a non-faulty mode, after which the 3 participants can communicate and remain *live* (Δ_{6-8}). The table on the right side of Fig. 8 gives $\mathbb{P}^{\min}_{\text{live}}(\Delta)$ and $\mathbb{P}^{\min}_{\text{df}}(\Delta)$ for each Δ. Worthy of note is the probability of *liveness* is 0 in Δ_{10-11}, as there exists a *fair* path (retry indefinitely) that starves u — a fair adversarial scheduler can always pick the action $\Delta_{10} \xrightarrow{\mathsf{s}:\tau}_n \Delta_{11}$, keeping the region closed. The full type of Δ_1 is given in [36, Ex. C.1].

Example 4.17 (Separating examples). Fig. 9 summarises the relationships between the properties of this section by exhibiting a few examples that fall into different categories of either *partially* or *fully* satisfying a property. The examples are numbered to match their regions in the diagram.

The properties are stable along *subtyping*: replacing a context by any subtype — as the verification workflow of Fig. 1 does when checking an inferred context against a declared one — preserves every property of this section, qualitatively and in probability:

Proposition 4.18 ($\leqslant$-monotonicity). *Let* $\Delta \leqslant \Delta'$ *and* $\mathsf{safe}(\Delta')$, *then:*
(1) $\varphi(\Delta') \implies \varphi(\Delta)$ *if* $\varphi \in \{\mathsf{safe}, \mathsf{df}, \mathsf{live}\}$*; and (2)* $\mathbb{P}^{\min}_{\text{df}}(\Delta') \leq \mathbb{P}^{\min}_{\text{df}}(\Delta)$ *and* $\mathbb{P}^{\min}_{\text{live}}(\Delta') \leq \mathbb{P}^{\min}_{\text{live}}(\Delta)$.

Thus, properties model checked for Δ' transfer to every subtype and, by [T-SUB], to every session typed below it. On the other hand, *widening* to a supertype may weaken the bounds by admitting additional behaviours: the guarantees remain sound but need not be tight.

Example 4.19 (Bounds). To show how $\leqslant$ may weaken bounds, consider first the session:

$$\vdash (\mathsf{q} \triangleleft \mathsf{flip}(0.5)\{\mathrm{H} \Rightarrow \mathsf{p}!\mathsf{m} \parallel \mathrm{T} \Rightarrow \mathsf{r}!\mathsf{m}\} \mid \mathsf{p} \triangleleft \mathsf{q}?\mathsf{m}) : \mathsf{q} : \oplus\{0.5 : \mathsf{p}!\mathsf{m}, 0.5 : \mathsf{r}!\mathsf{m}\},\ \mathsf{p} : \mathsf{q}?\mathsf{m}$$

whose type is partially deadlock-free and live as shown in bullet 4 of Fig. 9. In this case the session and typing context LTS are isomorphic, and indeed their lower bounds are identical. Consider now what happens when q has an always-true *if*-statement over action p!m and its previous *coinflip*:

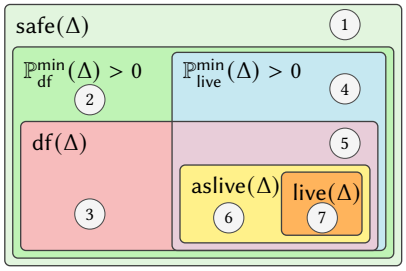
$$\begin{aligned} &\vdash \quad \mathsf{q} \triangleleft \text{if true then } \mathsf{p}!\mathsf{m} \text{ else } \mathsf{flip}(0.5)\{\mathrm{H} \Rightarrow \mathsf{p}!\mathsf{m} \parallel \mathrm{T} \Rightarrow \mathsf{r}!\mathsf{m}\} \mid \mathsf{p} \triangleleft \mathsf{q}?\mathsf{m} \\ &: \quad \mathsf{q} : \Sigma\{\oplus\{\mathsf{p}!\mathsf{m}\},\ \oplus\{0.5 : \mathsf{p}!\mathsf{m}, 0.5 : \mathsf{r}!\mathsf{m}\}\},\ \mathsf{p} : \mathsf{q}?\mathsf{m} \end{aligned}$$

By [SUB-Σ] the two branches have been widened to a nondeterministic choice (abstracting away the conditional used to make the choice), but the cost is obvious; while the session is clearly *live* the type still reports probability 0.5 of satisfying *liveness*, as an adversarial scheduler may always choose to pick the coinflip branch.

Finally, we note merging is also compatible with refinement:

Lemma 4.20 ($\sqcap$-monotonicity). *Suppose* $\mathsf{T}_i \leqslant \mathsf{T}'_i$ *for* $i \in \{1, 2\}$ *and* $\mathsf{T}'_1 \sqcap_p \mathsf{T}'_2$ *is defined, then* $\mathsf{T}_1 \sqcap_p \mathsf{T}_2$ *is defined and* $\mathsf{T}_1 \sqcap_p \mathsf{T}_2 \leqslant \mathsf{T}'_1 \sqcap_p \mathsf{T}'_2$.

Hence, replacing its operands by subtypes can only strengthen the resulting bounds.



	Examples
1	$\mathtt{p} : \oplus\mathtt{q}!\mathrm{m}, \quad \mathtt{q} : \oplus\mathtt{p}!\mathrm{m}$
2	$\mathtt{p} : \mu\mathtt{t}.\&\{\mathtt{q}?\mathrm{m}.\mathtt{t}\}, \quad \mathtt{r} : \oplus\mathtt{q}!\mathrm{m}, \quad \mathtt{q} : \oplus\{0.5 : \mathtt{r}!\mathrm{m}, \ 0.5 : \mathtt{p}!\mathrm{m}.\mu\mathtt{t}.\oplus\mathtt{p}!\mathrm{m}.\mathtt{t}\}$
3	$\mathtt{p} : \mu\mathtt{t}.\&\{\mathtt{q}?\mathrm{m}.\mathtt{t}\}, \quad \mathtt{q} : \mu\mathtt{t}.\oplus\mathtt{p}!\mathrm{m}.\mathtt{t}, \quad \mathtt{r} : \&\mathtt{p}?\mathrm{m}$
4	$\mathtt{p} : \&\{\mathtt{q}?\mathrm{m}\}, \quad \mathtt{q} : \oplus\{0.5 : \mathtt{p}!\mathrm{m}, \ 0.5 : \mathtt{r}!\mathrm{m}\}$
5	$\mathtt{p} : \mu\mathtt{t}.\&\{\mathtt{q}?\mathrm{m}.\mathtt{t}, \ \mathtt{r}?\mathrm{m}.\mathtt{t}\}, \quad \mathtt{q} : \mu\mathtt{t}.\oplus\mathtt{p}!\mathrm{m}.\mathtt{t}, \quad \mathtt{r} : \oplus\{0.5 : \mathtt{p}!\mathrm{m}, \ 0.5 : \mathtt{q}!\mathrm{m}\}$
6	$\mathtt{p} : \mu\mathtt{t}.\oplus\{0.5 : \mathtt{q}!\mathrm{m}.\mathtt{t}, \ 0.5 : \mathtt{r}!\mathrm{m}.\mathtt{t}\}, \quad \mathtt{q} : \mu\mathtt{t}.\&\{\mathtt{p}?\mathrm{m}.\mathtt{t}\}, \quad \mathtt{r} : \mu\mathtt{t}.\&\{\mathtt{p}?\mathrm{m}.\mathtt{t}\}$
7	$\mathtt{p} : \oplus\mathtt{q}!\mathrm{m}, \quad \mathtt{q} : \&\mathtt{p}?\mathrm{m}$

Fig. 9. Relationships between the properties. Numbers identify examples in the right-hand panel which separate properties. For example, the typing context in 2 is safe and partially deadlock-free, but not live nor completely deadlock-free.

4.4 Probabilistic Deadlock-Freedom and Liveness for Multiparty Processes

In this section we transfer the bounds obtained in the previous section on Δ back to sessions M typed as $\vdash M : \Delta$. We first adapt the definitions of §4.2 and 4.3 from typing contexts to sessions, and then show that $\mathbb{P}^{\min}_{\mathrm{df}}(\Delta)$ and $\mathbb{P}^{\min}_{\mathrm{live}}(\Delta)$ bound the corresponding properties of M.

Definition 4.21 (Session paths). A (session) path ξ is a sequence $M_1; M_2; \cdots$ with $M_i \longrightarrow_{p_i} M_{i+1}$ for all $i \in [1, |\xi|)$; *complete* if infinite or $M_{|\xi|} \not\to$.

Definition 4.22 (Session scheduler). A scheduler $\phi \in \Phi^M$ for session M is a function mapping each finite path $M_1; \cdots; M_n$ of M with $M_n \to$ to a participant $\mathtt{p}$ such that if $\phi(M_1; \cdots; M_n) = \mathtt{p}$, then either $M_n \Rrightarrow \mathtt{p} \triangleleft P \mid M'$ and $\mathtt{p} \triangleleft P \longrightarrow_p \mathtt{p} \triangleleft P'$ for some p, or $M_n \xrightarrow{(\mathtt{p},\mathtt{q})\ell}_1$ for some $\mathtt{q}$ and ℓ.

Complete paths $\Xi(M)$ and $\Xi_\phi(M)$, and probability measure $\mathbb{P}^\phi_M$ are as defined in Defs. 4.6 and 4.7.

Definition 4.23 (Fair session paths). A session path ξ is *fair*, written $\mathrm{fair}(\xi)$ if:

(1) $\forall \mathtt{p}, \mathtt{q}$: if $\forall n\, \exists m \geq n, \ell$ with $M_m \xrightarrow{(\mathtt{p},\mathtt{q})\ell}$, then $\forall n\, \exists m \geq n$ with $M_m \xrightarrow{(\mathtt{p},\mathtt{q})\ell} M_{m+1}$
(2) $\forall \mathtt{p}\, \forall n$: if $M_n \Rrightarrow \mathtt{p} \triangleleft \mathrm{flip}(p)\{\mathrm{H} \Rightarrow P \parallel \mathrm{T} \Rightarrow Q\} \mid M'$ or $M_n \Rrightarrow \mathtt{p} \triangleleft \mathrm{if}\ \mathrm{e}\ \mathrm{then}\ P\ \mathrm{else}\ Q \mid M'$, then $\exists m > n$ with $M_m \Rrightarrow \mathtt{p} \triangleleft P \mid M''$ or $M_m \Rrightarrow \mathtt{p} \triangleleft Q \mid M''$

Definition 4.24 (Live session paths). A session path ξ is *live*, written $\mathrm{live}(\xi)$ if for every $M_n \in \xi$:

(1) if $M_n \Rrightarrow \mathtt{p} \triangleleft \mathtt{q}!\ell\langle \mathrm{e} \rangle.P \mid M'$, then $\exists m \geq n$ with $M_m \xrightarrow{(\mathtt{p},\mathtt{q})\ell} M_{m+1}$
(2) if $M_n \Rrightarrow \mathtt{q} \triangleleft \sum_{i \in I} \mathtt{p}_i?\ell_i(x_i).Q_i \mid M'$, then $\exists m \geq n, j \in I$ with $M_m \xrightarrow{(\mathtt{p}_j,\mathtt{q})\ell_j} M_{m+1}$

A session M is *live*, written $\mathrm{live}(M)$, if every fair complete $\xi \in \Xi(M)$ is live; M is $\mathrm{deadlock}(M)$, if $M \not\to$ and $\exists \mathtt{p}, P : M \Rrightarrow \mathtt{p} \triangleleft P \mid M'$ with $P \neq \mathbf{0}$; and $\mathrm{df}(M)$ if $M \longrightarrow^* M'$ implies $\neg\mathrm{deadlock}(M')$.

Analogously to Def. 4.14, we define the *lower bounds* on these properties over all fair schedulers:

Definition 4.25 (Probabilistic properties of sessions).

$$\mathbb{P}^{\min}_{\mathrm{df}}(M) = 1 - \max_{\phi \in \Phi^M_F} \mathbb{P}^\phi_M(\Diamond\ \mathrm{deadlock}) \qquad \mathbb{P}^{\min}_{\mathrm{live}}(M) = \min_{\phi \in \Phi^M_F} \mathbb{P}^\phi_M(\mathrm{live})$$

with $\mathrm{asdf}(M)$ and $\mathrm{aslive}(M)$ as defined in Def. 4.15.

The main theorems of this paper follow; each property transfers from Δ to M, qualitatively and in probability.

Theorem 4.26. *If* $\vdash M : \Delta$, *then (1)* $\mathrm{df}(\Delta)$ *implies* $\mathrm{df}(M)$*; and (2)* $\mathbb{P}^{\min}_{\mathrm{df}}(\Delta) \leq \mathbb{P}^{\min}_{\mathrm{df}}(M)$

$$[\text{M-MODULE}]\ \frac{[\alpha]\, g \to \sum_{i\in I} p_i : u_i \in \{C_j\}_{j\in J}}{\{C_j\}_{j\in J} \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i : u_i} \qquad [\text{M-}\varepsilon]\ \frac{M \rightsquigarrow [\,]\, g \to \sum_{i\in I} p_i : u_i}{M \,||\, N \rightsquigarrow [\,]\, g \to \sum_{i\in I} p_i : u_i}$$

$$[\text{M-MOVE}]\ \frac{\alpha \neq \varepsilon \qquad M \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i : u_i \qquad \forall J, p'_j, u'_j, g' : N \not\rightsquigarrow [\alpha]\, g' \to \sum_{j\in J} p'_j : u'_j}{M \,||\, N \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i : u_i}$$

$$[\text{M-SYNC}]\ \frac{\alpha \neq \varepsilon \quad M \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i : u_i \quad N \rightsquigarrow [\alpha]\, g' \to \sum_{j\in J} p'_j : u'_j}{M \,||\, N \rightsquigarrow [\alpha]\, g \wedge g' \to \sum_{i,j} p_i \times p'_j : u_i \wedge u'_j} \qquad [\text{S-STEP}]\ \frac{M \rightsquigarrow [\alpha]\, g \to \sum_{i\in I} p_i : u_i \quad S \vdash g}{(M,S) \xrightarrow{\alpha}_{p_i} (M, S[u_i])}$$

Fig. 10. Rules for $\rightsquigarrow$ ([M-MODULE, ε, MOVE, SYNC]) and for model-state pairs ([S-STEP])

Theorem 4.27 (Liveness). *If* $\vdash M : \Delta$, *then (1)* $\mathsf{live}(\Delta)$ *implies* $\mathsf{live}(M)$; *(2)* $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta) \leq \mathbb{P}^{\min}_{\mathsf{live}}(M)$.

Corollary 4.28. *If* $\vdash M : \Delta$, *then* $\mathsf{asdf}(\Delta)$ *implies* $\mathsf{asdf}(M)$ *and* $\mathsf{aslive}(\Delta)$ *implies* $\mathsf{aslive}(M)$.

5 Translating Probabilistic Multiparty Session Types into PRISM

This section presents the translation from probabilistic multiparty session types to the PRISM modelling language. We first introduce the syntax and semantics of the PRISM language (§5.1), before describing the translation function (§5.2). Finally, we show the correctness of this encoding with respect to the operational semantics of session types and of PRISM (§5.3).

5.1 PRISM Syntax and Operational Semantics

To enable a precise discussion, we first formally define the relevant subset of the PRISM language. We closely follow [14, 57], with modifications to syntax and a more precise discussion of states.

Definition 5.1 (PRISM Syntax). The syntax of PRISM *models* is given by

$$\begin{aligned}
M, N &::= \{C_i\}_{i\in I} \mid M \,||\, N && \text{(module, composition)}\\
C &::= [\alpha]\, g \to \textstyle\sum_{i\in I} p_i : u_i && \text{(command, with } \textstyle\sum_{i\in I} p_i = 1 \text{ and } p_i \in [0,1])\\
g, g' &::= (x = v) \mid \neg g \mid g \wedge g' \mid g \vee g' \qquad u, u' ::= (x' = v) \mid u \wedge u' && \text{(guard, update)}\\
v, v' &::= v + v' \mid v - v' \mid v \times v' \mid v \div v' \mid x \mid n \in \mathbb{Z} && \text{(operations, variable, constant integer)}
\end{aligned}$$

PRISM models are a *composition* of *modules*, each containing *commands* C_i. Each command has an *action* α (potentially empty, denoted as $\alpha = \varepsilon$), *guard* g, and updates u_i, each with probability p_i. Guards and updates operate on *states* throughout the execution of the model.

Definition 5.2. A **state** S is a mapping $\{x_i \mapsto n_i\}_{i\in I}$ from variables x_i to constants n_i. **Updates** override mappings: $S[(x' = n)] = (S \setminus \{x \mapsto S(x)\}) \cup \{x \mapsto n\}$. PRISM distinguishes equality from updates by denoting them as $(x = v)$ and $(x' = v)$, respectively: in the latter, we update x, not x'. A state S is **a state for** model M if every $x \in \mathrm{dom}(S)$ appears in some guard or update in M. We call (M, S) a **model-state pair**.

Notation 5.3. For tidiness, we treat zero- and false-valued variables differently: (1) for all $x \notin \mathrm{dom}(S)$, we treat $S(x) = 0 = \mathrm{false}$; and (2) $S \cup \{x \mapsto 0\} \equiv S \cup \{x \mapsto \mathrm{false}\} \equiv S$. Hence we succinctly describe the state $\{x \mapsto 42, y \mapsto 0\}$ as $\{x \mapsto 42\}$. Here, we initialise integer variables to 0 and Booleans to false, so the *initial state* of any model is $\varnothing$. We write M as a shorthand for $(M, \varnothing)$.

We now define the semantics of the PRISM language. Following [14], we first introduce a relation $M \rightsquigarrow C$ (pronounced *"M has command C"*) to capture the effects of composition. The rules for $\rightsquigarrow$ are shown in Fig. 10. [M-MODULE] says a model with one module inherits its commands, while [M-ε] ensures that commands with empty actions are preserved. If M has action α and N does not,

[M-MOVE] keeps M's command in their composition $M \mid\mid N$. If both contain α, [M-SYNC] synchronises them, executing both commands together. Based on this, we define the transition relation $\xrightarrow{\alpha}_p$ between model–state pairs (Fig. 10). Its sole rule [S-STEP] does what we expect: if M has a command guarded by g, and state S satisfies g ($S \vdash g$), then with probability p_i, we apply the update u_i. Since each probabilistic branch becomes a separate transition, we assume that no two enabled commands with more than one branch share an action, so that the distributions remain recoverable from $\xrightarrow{\alpha}_p$. This is guaranteed by our encoding (§ 5.2.2). Similarly to typing contexts, we write $\rightarrow_p^*$ for the (probabilistic) reflexive, transitive closure.

5.2 Translating Multiparty Session Type Semantics

5.2.1 Dealing with States. The main complexity for the encoding arises from the fact that session types are regarded as a term rewriting system, whereas PRISM is *state-based*. Thus, we must first determine how to map terms into states. To facilitate this, we will be translating each participant in the typing context into its own module, with variable $S_{\mathtt{p}}$ tracking $\mathtt{p}$'s current position in its type. We then compose these modules into a single model.

We inductively define the *state space* of a type T, written $\mathrm{SS}(\mathsf{T})$, as: $\mathrm{SS}(\mathsf{end}) = 0$; $\mathrm{SS}(\mu \mathbf{t}.\mathsf{T}') = \mathrm{SS}(\mathsf{T}')$; $\mathrm{SS}(\mathbf{t}) = 0$; $\mathrm{SS}(\&_{i\in I}\ \mathtt{p_i}?\ell_i(\mathsf{B}_i).\mathsf{T}_i) = 1 + \sum_{i\in I} \mathrm{SS}(\mathsf{T}_i)$; $\mathrm{SS}(\oplus\, 1{:}\,\mathtt{q}!\ell\langle\mathsf{B}\rangle.\mathsf{T}') = 1 + \mathrm{SS}(\mathsf{T}')$ for a *committed* selection (a single branch of probability 1); for a proper selection ($|J| \geq 2$), $\mathrm{SS}(\oplus_{j\in J}\, p_j{:}\,\mathtt{p_j}!\ell_j\langle\mathsf{B}_j\rangle.\mathsf{T}_j) = 1 + \sum_{j\in J} \mathrm{SS}(\mathsf{T}^!_j)$, where $\mathsf{T}^!_j = \oplus\, 1{:}\,\mathtt{p_j}!\ell_j\langle\mathsf{B}_j\rangle.\mathsf{T}_j$ denotes the committed selection for branch j; and, for a proper sum ($|I| \geq 2$), $\mathrm{SS}(\sum_{i\in I}\oplus_{j\in J_i} p_{(i,j)}{:}\,\mathtt{p_j}!\ell_j\langle\mathsf{B}_j\rangle.\mathsf{T}_{(i,j)}) = 1 + \sum_{i\in I} \mathrm{SS}(\mathsf{T}^{\oplus}_i)$, where $\mathsf{T}^{\oplus}_i = \oplus_{j\in J_i}\, p_{(i,j)}{:}\,\mathtt{p_j}!\ell_j\langle\mathsf{B}_j\rangle.\mathsf{T}_{(i,j)}$ denotes the i-th summand of the sum (a singleton selection). Each variable $S_{\mathtt{p}}$ in the encoding of $\Delta = \mathtt{p} : \mathsf{T}, \Delta'$ has a value between 0 and $\mathrm{SS}(\mathsf{T})$, inclusive, where $S_{\mathtt{p}} = \mathrm{SS}(\mathsf{T})$ represents the end state.

The counts mirror the type-level semantics, which resolves the nondeterministic choice between summands, the probabilistic choice between branches, and the communication itself by separate reductions. A branching needs only an entry state, from which the participant synchronises with the chosen sender. A committed selection similarly needs only an entry state, from which the participant synchronises with the recipient. A proper selection contributes an entry state, from which the participant commits to a branch probabilistically, moving to the entry state of the corresponding committed selection $\mathsf{T}^!_j$. A proper sum contributes an entry state from which a summand is picked nondeterministically. Recursion contributes no states, as we simply loop back to the state of the corresponding μ-binding. Similarly, end also contributes no states, since all terminated participants share the single end state. The transitions between these states are made precise by the encoding below.

5.2.2 Encoding. We define an encoding from typing contexts into PRISM. We first present an *inner* encoding $(\!|\cdot|\!)$, and then extend this with the *closure module* to give the final encoding $[\![\cdot]\!]$.

Inner top-level encoding $(\!|\cdot|\!)$. A typing context Δ is translated as: $(\!|\varnothing|\!) = \varnothing$; $(\!|\Delta, \Delta'|\!) = (\!|\Delta|\!) \mid\mid (\!|\Delta'|\!)$, and $(\!|\mathtt{p} : \mathsf{T}|\!) = \{\!|\mathsf{T}|\!\}_{(\mathtt{p},0,\mathrm{SS}(\mathsf{T}),\varnothing)}$ with the type-level encoding $\{\!|\mathsf{T}|\!\}_{(\mathtt{p},n,m,f)}$, which takes a type T being translated, participant $\mathtt{p}$, current state n, end state m and a mapping f from variables to their state value.

Inner type-level encoding $\{\!|\cdot|\!\}_{(\mathtt{p},n,m,f)}$. The inner type-level encoding is presented in Fig. 11. Each reduction of the typing context LTS maps to exactly *one* PRISM transition, distinguished by its action label. For a proper sum ($|I| \geq 2$), the nondeterministic reduction $\xrightarrow{\mathtt{p}:\tau}_n$ maps to an internal transition $\xrightarrow{\mathtt{p}::\tau}_1$ (mirroring the type-level action $\mathtt{p}{:}\tau$) that picks a summand $i \in I$ and moves to the entry state of the encoding of the singleton selection $\mathsf{T}^{\oplus}_i$. For a proper selection, the probabilistic

reduction $\xrightarrow{\mathsf{p}:p}_p$ (rule [CT-PROB]) maps to an internal transition $\xrightarrow{\mathsf{p}::p}_p$ in which p probabilistically picks a branch $j \in J$ and moves to the entry state of the encoding of the committed selection T'_j. For a committed selection, the communication $\xrightarrow{(\mathsf{p},\mathsf{q})\ell}_1$ maps to the synchronisation $\xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_1$ that informs q of p's choice. Since participant names are unique, no other module has an action labelled $\mathsf{p} :: \tau$ or $\mathsf{p} :: p$, so these transitions cannot synchronise and are therefore internal. For a branching, the receiver synchronises directly with the chosen sender in a single $\xrightarrow{\mathsf{q}::\mathsf{p}::\ell}_1$ transition. For recursion $\mu\mathbf{t}.\mathsf{T}$, we extend the mapping with $\mathbf{t} \mapsto n$ and continue translating T (using f if a continuation is a variable). We refer to the two kinds of internal transitions produced by the encoding as follows: a

$$\{\!|\mathsf{end}|\!\}_{(\mathsf{p},n,m,f)} = \varnothing \qquad \{\!|\mathbf{t}|\!\}_{(\mathsf{p},n,m,f)} = \varnothing \qquad \{\!|\mu\mathbf{t}.\mathsf{T}|\!\}_{(\mathsf{p},n,m,f)} = \{\!|\mathsf{T}|\!\}_{(\mathsf{p},n,m,f\cup\{\mathbf{t}\mapsto n\})}$$

$$\{\!|\textstyle\sum_{i\in I}\bigoplus_{j\in J_i} p_{(i,j)} : \mathsf{p}_\mathsf{j}!\ell_\mathsf{j}\langle \mathsf{B}_j\rangle.\mathsf{T}_{(i,j)}|\!\}_{(\mathsf{p},n,m,f)}$$
$$= \textstyle\bigcup_{i\in I} [\mathsf{p} :: \tau]\,(S_\mathsf{p} = n) \to 1 : (S'_\mathsf{p} = \mathsf{sel}(i)) \;\cup\; \bigcup_{i\in I}\{\!|\mathsf{T}^\oplus_i|\!\}_{(\mathsf{p},\mathsf{sel}(i),m,f)}$$
$$\text{where} \quad |I| \geq 2 \quad \mathsf{T}^\oplus_i = \textstyle\bigoplus_{j\in J_i} p_{(i,j)} : \mathsf{p}_\mathsf{j}!\ell_\mathsf{j}\langle \mathsf{B}_j\rangle.\mathsf{T}_{(i,j)} \quad \mathsf{sel}(i) = n + 1 + \sum_{i'<i} \mathsf{SS}(\mathsf{T}^\oplus_{i'})$$

$$\{\!|\textstyle\bigoplus_{j\in J} p_j : \mathsf{p}_\mathsf{j}!\ell_\mathsf{j}\langle \mathsf{B}_j\rangle.\mathsf{T}_j|\!\}_{(\mathsf{p},n,m,f)} \qquad (|J| \geq 2)$$
$$= [\mathsf{p} :: p]\,(S_\mathsf{p} = n) \to \textstyle\sum_{j\in J}(p_j : (S'_\mathsf{p} = \mathsf{com}(j))) \;\cup\; \bigcup_{j\in J}\{\!|\mathsf{T}'_j|\!\}_{(\mathsf{p},\mathsf{com}(j),m,f)}$$
$$\text{where} \quad \mathsf{T}'_j = \textstyle\bigoplus 1 : \mathsf{p}_\mathsf{j}!\ell_\mathsf{j}\langle \mathsf{B}_j\rangle.\mathsf{T}_j \quad \mathsf{com}(j) = n + 1 + \sum_{j'<j} \mathsf{SS}(\mathsf{T}'_{j'})$$

$$\{\!|\textstyle\bigoplus 1 : \mathsf{q}!\ell\langle \mathsf{B}\rangle.\mathsf{T}'|\!\}_{(\mathsf{p},n,m,f)}$$
$$= [\mathsf{p} :: \mathsf{q} :: \ell]\,(S_\mathsf{p} = n) \to 1 : \begin{cases} S'_\mathsf{p} = m & \mathsf{T}' = \mathsf{end} \\ S'_\mathsf{p} = f(\mathbf{t}) & \mathsf{T}' = \mathbf{t} \\ S'_\mathsf{p} = n+1 & \text{otherwise} \end{cases} \cup\; \{\!|\mathsf{T}'|\!\}_{(\mathsf{p},n+1,m,f)}$$

$$\{\!|\textstyle\&_{i\in I}\ \mathsf{p}_\mathsf{i}?\ell_\mathsf{i}(\mathsf{B}_i).\mathsf{T}_i|\!\}_{(\mathsf{p},n,m,f)}$$
$$= \textstyle\bigcup_{i\in I}[\mathsf{p}_\mathsf{i} :: \mathsf{p} :: \ell_i]\,(S_\mathsf{p} = n) \to 1 : \begin{cases} S'_\mathsf{p} = m & \mathsf{T}_i = \mathsf{end} \\ S'_\mathsf{p} = f(\mathbf{t}) & \mathsf{T}_i = \mathbf{t} \\ S'_\mathsf{p} = \mathsf{next}(i) & \text{otherwise} \end{cases} \cup \bigcup_{i\in I}\{\!|\mathsf{T}_i|\!\}_{(\mathsf{p},\mathsf{next}(i),m,f)}$$
$$\text{where} \quad \mathsf{next}(i) = n + 1 + \textstyle\sum_{i'<i} \mathsf{SS}(\mathsf{T}_{i'})$$

Fig. 11. Inner Translation

summand step is a transition labelled $\mathsf{p} :: \tau$, introduced for a proper sum, moving S_p from the entry state of the sum to the entry state of one of its summands $\mathsf{T}^\oplus_i$ with probability 1; a *commitment step* is a transition labelled $\mathsf{p} :: p$, introduced for a proper selection, moving S_p from the entry state of the selection to the entry state of one of its committed selections T'_j with probability p_j. Summand steps resolve the nondeterministic choice (rule [CT-ND]), commitment steps resolve the probabilistic choice (rule [CT-PROB]), and synchronisations perform the communication (rule [CT-τ]). Commitment steps are the only commands with more than one branch, and at most one is enabled in any state per module, as each is guarded by a distinct entry state and never synchronises.

The inner encoding has slightly more transitions than needed since we treat each $\xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$ transition as a communication, implicitly assuming both p and q participate. However, in some cases, only one of p or q contains this action, and [M-MOVE] permits such transitions to occur without synchronisation. Thus, we introduce a *closure module* that blocks these spurious transitions.

Definition 5.4. Let $\mathsf{actions}_\Delta(\mathsf{p})$ be the set of actions in $(\!|\mathsf{p} : \Delta(\mathsf{p})|\!)$ where $\mathsf{actions}_\Delta(\mathsf{p}) = \varnothing$ if $\mathsf{p} \notin \mathsf{dom}(\Delta)$. Define $\mathsf{actions}(\Delta) = \bigcup_{\mathsf{p}\in\mathsf{dom}(\Delta)} \mathsf{actions}_\Delta(\mathsf{p})$. The **closure module** for Δ is given by

$$\mathsf{closure}(\Delta) = \{[\mathsf{p} :: \mathsf{q} :: \ell]\ \mathsf{false} \to 1 : () \mid \mathsf{p} :: \mathsf{q} :: \ell \in \mathsf{actions}_\Delta(\mathsf{p}) \oplus \mathsf{actions}_\Delta(\mathsf{q})\},$$

$$
(\!|\mathbf{c} : \mathsf{T}_\mathbf{c}|\!) = \begin{cases} [\mathbf{s} :: \mathbf{c} :: \mathsf{incr}]\ (S_\mathbf{c} = 0) \longrightarrow 1 : (S'_\mathbf{c} = 1) \\ [\mathbf{c} :: \tau]\ (S_\mathbf{c} = 1) \longrightarrow 1 : (S'_\mathbf{c} = 2), \\ [\mathbf{c} :: \tau]\ (S_\mathbf{c} = 1) \longrightarrow 1 : (S'_\mathbf{c} = 3), \\ [\mathbf{c} :: \mathbf{u} :: \mathsf{res}]\ (S_\mathbf{c} = 2) \longrightarrow 1 : (S'_\mathbf{c} = 0), \\ \quad [\mathbf{c} :: p]\ (S_\mathbf{c} = 3) \longrightarrow \quad 0.8 : (S'_\mathbf{c} = 4) \\ \qquad\qquad\qquad\quad + \quad 0.2 : (S'_\mathbf{c} = 5), \\ [\mathbf{c} :: \mathbf{u} :: \mathsf{res}]\ (S_\mathbf{c} = 4) \longrightarrow 1 : (S'_\mathbf{c} = 0), \\ [\mathbf{c} :: \mathbf{s} :: \mathsf{error}]\ (S_\mathbf{c} = 5) \longrightarrow 1 : (S'_\mathbf{c} = 0)\ \} \end{cases}
$$

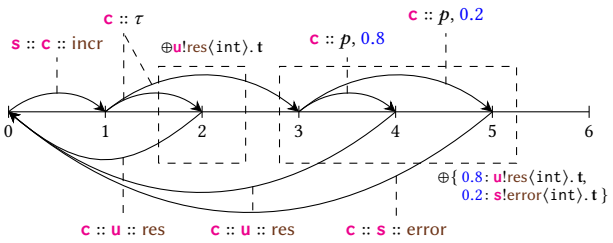


Fig. 12. States in the encoding of $\mathbf{c} : \mathsf{T}_\mathbf{c}$ (Example 3.1). Points on the number line represent values of $S_\mathbf{c}$, and arrows represent transitions, labelled with their action and probability. Unlabelled transitions have $p = 1$.

where $\oplus$ is the *symmetric difference*, and the $()$ denotes a dummy update that will never be applied. The **final encoding** $[\![\cdot]\!]$ of contexts into PRISM models is defined by $[\![\Delta]\!] = (\!|\Delta|\!) \mid\mid \mathsf{closure}(\Delta)$.

Example 5.5. Consider the calculator $\mathbf{c} : \mathsf{T}_\mathbf{c}$ from Example 3.1, for which $\mathsf{SS}(\mathsf{T}_\mathbf{c}) = 6$. Fig. 12 shows its state encoding. Each typing context reduction corresponds to exactly *one* PRISM transition. The recursion contributes no state, so the branching $\&\mathbf{s}?\mathsf{incr}$ is translated at state 0 with $f = \{\mathbf{t} \mapsto 0\}$. Then, receiving incr from $\mathbf{s}$ moves $S_\mathbf{c}$ to state 1, which is the entry state of the nondeterministic sum. From there, a summand step picks one of the two summands nondeterministically, moving either to state 2, the singleton selection $\oplus\mathbf{u}!\mathsf{res}.\mathbf{t}$, or to state 3, the entry state of the probabilistic selection. From state 3, a commitment step moves to state 4 with probability 0.8 or to state 5 with probability 0.2, the entry states of the two singleton selections $\oplus\mathbf{u}!\mathsf{res}.\mathbf{t}$ and $\oplus\mathbf{s}!\mathsf{error}.\mathbf{t}$. Each singleton selection then synchronises with its recipient. As every continuation is the variable $\mathbf{t}$, these synchronisations loop back to state $f(\mathbf{t}) = 0$. The end state 6 is never reached, since the calculator never terminates.

The remaining participants $\mathbf{u}$ and $\mathbf{s}$ are encoded in a similar manner, yielding modules over $S_\mathbf{u}$ and $S_\mathbf{s}$. The final encoding is then: $[\![\Delta]\!] = (\!|\mathbf{c} : \mathsf{T}_\mathbf{c}|\!) \mid\mid (\!|\mathbf{u} : \mathsf{T}_\mathbf{u}|\!) \mid\mid (\!|\mathbf{s} : \mathsf{T}_\mathbf{s}|\!) \mid\mid \mathsf{closure}(\Delta)$

The resulting modules synchronise on the four actions $\mathbf{u} :: \mathbf{s} :: \mathsf{incr}$, $\mathbf{s} :: \mathbf{c} :: \mathsf{incr}$, $\mathbf{c} :: \mathbf{u} :: \mathsf{res}$ and $\mathbf{c} :: \mathbf{s} :: \mathsf{error}$. Every action appears in both the sender's and the receiver's module, so the closure module is empty. Tracing part of an execution, we could have:

$$
([\![\Delta]\!], \{S_\mathbf{c} \mapsto 0, \ldots\}) \xrightarrow{\mathbf{s}::\mathbf{c}::\mathsf{incr}}_1 ([\![\Delta]\!], \{S_\mathbf{c} \mapsto 1, \ldots\}) \xrightarrow{\mathbf{c}::\tau}_1 ([\![\Delta]\!], \{S_\mathbf{c} \mapsto 3, \ldots\}) \xrightarrow{\mathbf{c}::p}_{0.8} ([\![\Delta]\!], \{S_\mathbf{c} \mapsto 4, \ldots\})
$$

5.3 Correctness

We now show that the encoding is *sound* (Theorem 5.6): every typing context reduction corresponds to a transition of the encoded model; and *complete* (Theorem 5.7): every PRISM transition corresponds to a typing context reduction. Detailed definitions and proofs are given in [36, App. D].

5.3.1 Soundness. For soundness, we must show that $(\!|\cdot|\!)$ preserves behaviour under recursion unfolding: $(\!|\mathbf{p} : \mu\mathbf{t}.\mathsf{T}|\!)$ should be *equivalent* to $(\!|\mathbf{p} : \mathsf{T}\{^{\mu\mathbf{t}.\mathsf{T}}/_\mathbf{t}\}|\!)$. Though the two modules' internal states and commands may differ, they should exhibit the same observable transition behaviour. This motivates use of *bisimulation* on model-state pairs $(M_1, I_1) \sim (M_2, I_2)$ to define equivalence.

Theorem 5.6 (Translation soundness). *Let Δ, Δ' be contexts.*

(1) If $\Delta \xrightarrow{\mathbf{p}:\tau}_n \Delta'$, then there exists a state S such that $[\![\Delta]\!] \xrightarrow{\mathbf{p}::\tau}_1 ([\![\Delta]\!], S)$ and $[\![\Delta']\!] \sim ([\![\Delta]\!], S)$.

(2) If $\Delta \xrightarrow{\mathbf{p}:p}_p \Delta'$, then exists a state S such that $[\![\Delta]\!] \xrightarrow{\mathbf{p}::p}_p ([\![\Delta]\!], S)$ and $[\![\Delta']\!] \sim ([\![\Delta]\!], S)$.

(3) If $\Delta \xrightarrow{(\mathbf{p},\mathbf{q})\ell}_1 \Delta'$, then exists a state S such that $[\![\Delta]\!] \xrightarrow{\mathbf{p}::\mathbf{q}::\ell}_1 ([\![\Delta]\!], S)$ and $[\![\Delta']\!] \sim ([\![\Delta]\!], S)$.

5.3.2 Completeness. Completeness is the dual of soundness: every transition of the encoded model corresponds directly to a reduction of the typing context.

Theorem 5.7 (Translation completeness). *For every context* Δ*:*

(1) if $[[\Delta]] \xrightarrow{\mathsf{p}::\tau}_1 ([[\Delta]], S)$*, then there exists* Δ' *such that* $\Delta \xrightarrow{\mathsf{p}:\tau}_n \Delta'$ *and* $([[\Delta]], S) \sim [[\Delta']]$*;*

(2) if $[[\Delta]] \xrightarrow{\mathsf{p}::p}_p ([[\Delta]], S)$*, then there exists* Δ' *such that* $\Delta \xrightarrow{\mathsf{p}:p}_p \Delta'$ *and* $([[\Delta]], S) \sim [[\Delta']]$*;*

(3) if $[[\Delta]] \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}_1 ([[\Delta]], S)$*, then there exists* Δ' *such that* $\Delta \xrightarrow{(\mathsf{p},\mathsf{q})\ell}_1 \Delta'$ *and* $([[\Delta]], S) \sim [[\Delta']]$*.*

6 Model-Checking Probabilistic Multiparty Session Types in PRISM

With the encoding in place, we must now express the properties of §4.3 as logical formulae to enable model checking. We use Probabilistic LTL (PLTL), the linear-time fragment of PCTL*, to specify and model check properties. See [36, App. E] for proofs of statements in this section.

A brief introduction to PLTL. Due to space constraints, we opt for a brief presentation; a more principled discussion can be found in [6, 9]. The syntax of our fragment of PLTL is given by

$$\begin{array}{rll} \Phi ::= & \mathsf{Pmin}\,(\varphi) \mid \mathsf{Pmax}\,(\varphi) & \text{(lower- and upper-bound probability constructs)} \\ \varphi, \psi ::= & a \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi & \text{(atomic proposition, negation, conjunction, disjunction)} \\ \mid & \Box\,\varphi \mid \Diamond\,\varphi & \text{(always, eventually)} \end{array}$$

We write $\varphi \implies \psi$ as a shorthand for $\neg\varphi \vee \psi$. We omit the *next* (X) and *until* (U) operators of PLTL, as they are not needed to express our properties.

Path π of a PRISM model M is a (potentially infinite) sequence of states $S_1; S_2; \cdots$ such that $\forall i \in [1, |\pi|) : (M, S_i) \to (M, S_{i+1})$. We call φ a *path formula*. Path formulae act on paths π of a model M, and we write $\pi \models \varphi$ if π satisfies φ. The semantics of path formulae are given in Def. 6.1.

Given a model M, $\mathsf{Pmin}\,(\varphi)$ (resp. $\mathsf{Pmax}\,(\varphi)$) tells us the **minimum** (resp. **maximum**) probability of obtaining a path π of M such that $\pi \models \varphi$, across all possible schedulers. Since our PRISM models are MDPs, they contain both probabilistic choice (within the same action) and nondeterminism (over which action to fire, when multiple are available).

Definition 6.1 (Semantics of a path formula). Let $\pi = S_1; S_2; \cdots$ be a path, and denote by $\pi[i..]$ the subpath $S_i; S_{i+1}; \cdots$. The *semantics of a path formula* φ is given inductively:

$$\pi \models a \iff S_1 \vdash a \qquad \pi \models \varphi \wedge \psi \iff \pi \models \varphi \wedge \pi \models \psi \qquad \pi \models \varphi \vee \psi \iff \pi \models \varphi \vee \pi \models \psi$$
$$\pi \models \neg\varphi \iff \pi \not\models \varphi \qquad \pi \models \Box\,\varphi \iff \forall i \cdot \pi[i..] \models \varphi \qquad \pi \models \Diamond\,\varphi \iff \exists i \cdot \pi[i..] \models \varphi$$

The meaning of $S_1 \vdash a$ will be explained in §6.1. The *always* and *eventually* operators originate from *linear temporal logic*; informally, $\Box\,\varphi$ is true if φ is always true throughout the path, and $\Diamond\,\varphi$ is true if φ eventually becomes true. Other operators are standard.

6.1 Defining Specifications

We define several PRISM *labels* – boolean expressions over state variables – to act as atomic propositions. For conciseness, we only give informal meanings here, though it is straightforward to express them as conjunctions or disjunctions of state variables.

Definition 6.2. Given an encoding $[[\Delta]]$, we define the *labels* (their names denoted with *"quotes"*):

- $S \vdash$ *"end"* $\iff \forall \mathsf{p} \in \mathrm{dom}(\Delta) : S(S_\mathsf{p}) = \mathrm{SS}(\Delta(\mathsf{p}))$
- $\forall \mathsf{p} :: \mathsf{q} :: \ell \in \mathrm{actions}_\Delta(\mathsf{p}) \cup \mathrm{actions}_\Delta(\mathsf{q}) : S \vdash$ *"send* $:: \mathsf{p} :: \mathsf{q} :: \ell$*"* $\iff (\lfloor \Delta(\mathsf{p}) \rfloor, S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$
- $\forall \mathsf{p}, \mathsf{q} \in \mathrm{dom}(\Delta) : S \vdash$ *"receive* $:: \mathsf{p} :: \mathsf{q}$*"* $\iff \exists \ell : (\lfloor \Delta(\mathsf{q}) \rfloor, S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$
- $\forall \mathsf{p} :: \mathsf{q} :: \ell \in \mathrm{actions}_\Delta(\mathsf{p}) \cup \mathrm{actions}_\Delta(\mathsf{q}) : S \vdash$ *"receive* $:: \mathsf{p} :: \mathsf{q} :: \ell$*"* $\iff (\lfloor \Delta(\mathsf{q}) \rfloor, S) \xrightarrow{\mathsf{p}::\mathsf{q}::\ell}$

Additionally, PRISM has a built-in label *"deadlock"* such that $S \vdash$ *"deadlock"* $\iff ([[\Delta]], S) \not\to$.

The label *"end"* is true iff all participants are in the end state. *"send* :: p :: q :: ℓ*"* is true if p is able to send ℓ to q, i.e. p is at the entry state of a committed selection and offers the synchronisation p :: q :: ℓ; and *"receive* :: p :: q :: ℓ*"* is true if q is able to receive ℓ from p. Finally, *"receive* :: p :: q*"* is true if q is able to receive *any* label from p.

Fairness. The properties of Def. 4.14 are quantified over *fair* schedulers, whereas Pmin and Pmax quantify over *all* schedulers of the model. This restriction cannot be imposed directly, as PRISM applies fairness to all nondeterministic choices uniformly, including the internal nondeterminism $\Sigma_{i\in I}\mathsf{T}_i \to_n \mathsf{T}_k$, which must remain unconstrained. As we will soon see, we instead express deadlock-freedom and liveness through the reachability of their negations, for which the restriction to fair schedulers is unnecessary.

Safety and deadlock-freedom. Using the labels, we write the following *property specifications*:

$$\mathsf{safe}(\Delta) \iff \mathsf{Pmin}\big(\Box\,\big(\textstyle\bigwedge_{\mathsf{p}::\mathsf{q}::\ell\in\mathsf{actions}(\Delta)}\big(\text{"}send :: \mathsf{p} :: \mathsf{q} :: \ell\text{"} \wedge \text{"}receive :: \mathsf{p} :: \mathsf{q}\text{"}\big) \implies \text{"}receive :: \mathsf{p} :: \mathsf{q} :: \ell\text{"}\big)\big) = 1$$

$$\mathbb{P}^{\min}_{\mathsf{df}}(\Delta) = 1 - \mathsf{Pmax}\,(\Diamond\,(\text{"}deadlock\text{"} \wedge \neg\text{"}end\text{"})) \qquad \mathsf{asdf}(\Delta) \iff \mathbb{P}^{\min}_{\mathsf{df}}(\Delta) = 1$$

In a safe context (Def. 3.12), if p has committed to sending ℓ to q and q is ready to receive from p, then q must accept ℓ. For deadlock-freedom, recall from Def. 4.8 that a deadlocked context is one that cannot reduce even though some participant has not ended. The path formula $\Diamond\,(\text{"}deadlock\text{"} \wedge \neg\text{"}end\text{"})$ holds exactly on the paths that reach such a state, so the specification computes 1 minus the maximal probability of deadlock across all schedulers. The following theorem shows that this maximum agrees with the maximum over fair schedulers of Def. 4.14.

Theorem 6.3. $\max_{\phi\in\Phi^\Delta_F} \mathbb{P}^\phi_\Delta(\Diamond\ \mathsf{deadlock}) = \max_{\phi\in\Phi^\Delta} \mathbb{P}^\phi_\Delta(\Diamond\ \mathsf{deadlock})$

Intuitively, every deadlocking path is finite, and every finite complete path is fair. Admitting unfair schedulers therefore does not change the maximal probability of deadlock.

Liveness is trickier, as it is a property of infinite paths whose negation is not immediately a reachability event. We note that a context fails to be almost-surely live precisely when it can reach a region of states in which some participant can be kept pending forever while scheduling remains fair. We call this a *weak almost-sure livelock*, weak in that an actual livelock may only surface when the nondeterminism is resolved adversarially.

To define the *weak almost-sure livelock* region, we first define $\mathsf{Pend}_\mathsf{p}(\Delta)$ to be the set of states reachable from Δ where p is nonterminated but has no enabled communication:

$$\mathsf{Pend}_\mathsf{p}(\Delta) = \{\Delta' \in (\Delta\to^*) \mid \Delta'(\mathsf{p}) \neq \mathsf{end}, \forall \mathsf{q}, \ell, p :\ \Delta' \xrightarrow{(\mathsf{p},\mathsf{q})\ell} \!\!\!\!\!\!\to_p \wedge\ \Delta' \xrightarrow{(\mathsf{q},\mathsf{p})\ell} \!\!\!\!\!\!\to_p\}.$$

Writing Pend_p for membership in $\mathsf{Pend}_\mathsf{p}(\Delta)$, a fair path π is non-live iff $\pi \models \bigvee_{\mathsf{p}\in\mathsf{dom}(\Delta)} \Diamond\,\Box\,\mathsf{Pend}_\mathsf{p}$. Indeed, fairness resolves any remaining internal choices on such a pending suffix, leaving the participant permanently stuck.

Writing $\mathsf{visited}(\xi) = \{\Delta_i \mid \exists i \le |\xi| : \xi[i] = \Delta_i\}$ for the set of states visited along ξ, we define:

Definition 6.4 (Weak almost-sure livelock). The *weak almost-surely livelocked region* for p:

$$\mathsf{WASlivelock}_\mathsf{p}(\Delta) = \{\Delta' \in \mathsf{Pend}_\mathsf{p}(\Delta) \mid \exists \phi \in \Phi_F^{\Delta'} : \forall \xi \in \Xi_\phi(\Delta') : \mathsf{visited}(\xi) \subseteq \mathsf{Pend}_\mathsf{p}(\Delta)\}$$

and $\mathsf{WASlivelock}(\Delta) = \bigcup_{\mathsf{p}\in\mathsf{dom}(\Delta)} \mathsf{WASlivelock}_\mathsf{p}(\Delta)$.

For a *limit property* φ, the standard end-component reduction gives $\mathsf{Pmin}\,(\varphi) = 1 - \mathsf{Pmax}\,(\Diamond\,(\bigcup_{i\in I} S_i))$, where I indexes all end components (S_i, A_i) whose recurrent behaviour violates φ [6, Theorem 10.122(b)]. Applying this reduction with fairness incorporated yields:

Theorem 6.5. $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta) = \min_{\phi\in\Phi^\Delta_F} \mathbb{P}^\phi_\Delta(\mathsf{live}) = 1 - \max_{\phi\in\Phi^\Delta} \mathbb{P}^\phi_\Delta(\Diamond\ \mathsf{WASlivelock})$

Then, membership of the livelocked region becomes an atomic proposition. We define the label

$$S \vdash \textit{“livelock”} \iff \exists \Delta' \in \mathsf{WASlivelock}(\Delta) : ([[\Delta]], S) \sim [[\Delta']]$$

which, like the labels of Def. 6.2, is a finite disjunction over valuations of the state variables S_{p}. The implementation precomputes $\mathsf{WASlivelock}(\Delta)$ by an end-component analysis of the context MDP (details in [36, App. E.1]). By Theorem 6.5, we then obtain the liveness specification:

$$\mathbb{P}^{\min}_{\mathsf{live}}(\Delta) = 1 - \mathsf{Pmax}\,(\Diamond\ \textit{“livelock”})$$

Since the right-hand side is a reachability query maximised over all schedulers, no fairness constraint needs to be imposed on PRISM, and Δ is almost-surely live iff $\mathsf{Pmax}\,(\Diamond\ \textit{“livelock”}) = 0$.

Complexity. The runtime cost of our verification approach is dominated by the weak almost-sure livelock precomputation as well as the PRISM model-checking. Writing $|\Delta|$ for the total number of nodes in the syntax trees of the types in Δ, we have:

Theorem 6.6. *Let N be the number of reachable states of $[[\Delta]]$ and $R = |\{\Delta' \mid \Delta \rightarrow^{*} \Delta'\}|$. Then* $\mathsf{safe}(\Delta)$, $\mathbb{P}^{\min}_{\mathsf{df}}(\Delta)$ *and* $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta)$ *can be computed in time polynomial in N, R and $|\Delta|$.*

Corollary 6.7. *Let $k = |\,\mathrm{dom}(\Delta)|$. Then* $\mathsf{safe}(\Delta)$, $\mathbb{P}^{\min}_{\mathsf{df}}(\Delta)$ *and* $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta)$ *can be computed in time $|\Delta|^{O(k)}$, i.e. exponential in the number of participants.*

In practice, synchronous communication often tightly couples the reachable set, as can be seen in our empirical evaluation. For instance, the product of the state spaces of the ten participants of Multiparty Workers exceeds a million states, yet only 1222 of them are reachable (Table 1).

Example 6.8. We consider the calculator service in Example 3.1 for the final time, with the encoding given in Fig. 12. The label *“end”* requires all three state variables S_{u}, S_{s} and S_{c} to be at their end states, and, for instance, *“send* :: **c** :: **s** :: error*”* holds exactly when $S_{\mathsf{c}} = 5$ in Fig. 12, while *“receive* :: **c** :: **s***”* holds when S_{s} is at the entry state of the branching of T_{s}. PRISM reports that Δ is safe and that no reachable state satisfies *“deadlock”*, since the context never stops, and so $\mathbb{P}^{\min}_{\mathsf{df}}(\Delta) = 1$. The precomputed region $\mathsf{WASlivelock}(\Delta)$ is empty: **u** may be left pending while **s** and **c** exchange error and incr (Example 4.12), but no fair scheduler can keep all paths in this region, as every commitment step of **c** leaves it with probability 0.8. Hence *“livelock”* is the constant false and $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta) = 1$. The calculator is thus almost-surely deadlock-free and live.

For a negative case, consider the extended context Δ_1 of Example 4.16 (Fig. 8). Here PRISM finds exactly one state satisfying *“deadlock”* $\wedge \neg$ *“end”*, namely the encoding of Δ_9, in which S_{s} and S_{c} are at their end states while S_{u} is at the entry state of the branching &**c**?res. The label *“livelock”* is the disjunction of the valuations encoding Δ_9, where **u** is pending forever because other participants have ended, and Δ_{10} and Δ_{11}, where **u** is pending while **s** and **c** exchange retry. PRISM returns 0.31 and 0.474 for the two Pmax queries, giving $\mathbb{P}^{\min}_{\mathsf{df}}(\Delta_1) = 0.69$ and $\mathbb{P}^{\min}_{\mathsf{live}}(\Delta_1) = 0.526$.

6.2 Implementation and Performance Evaluation

We implement our encoding and property specifications in Prose, a tool for the verification of **pro**babilistic multiparty **se**ssion types. Prose has been implemented in roughly 2800 lines of OCaml. It accepts a typing context as its input, and outputs the probabilities of properties in §4.3. Prose is designed to be a self-contained, standalone toolkit for verifying types as abstractions of programs. Prose aims to be used in conjunction with real programs (such as Go or Scala programs) following the line of [7, 42, 53] which have applied model checkers for types to real programs.

We evaluate the efficiency of Prose through performance benchmarks. The experiments were run on a laptop with an Apple M4 processor and 16GB of RAM, with OCaml 5.2.1 and PRISM v4.10.1. We discard one warmup run, then repeat each measurement 30 times and report its mean and relative

Table 1. Benchmark results for Prose. WASL is the precomputation of the weak almost-sure livelocked region (Def. 6.4). All benchmarked processes can be found in the `/examples/` folder of the artifact [35].

	Mean runtime (in ms) of 30 runs after 1 warmup, ± relative standard error (%)							Model Size	
Typing context	**Inference**	**Translation**	**WASL**	**Safety**	**DF**	**Liveness**	**End-to-end**	**States**	**Trans.**
OAuth Protocol[1]	21.64±1.5%	0.42±4.8%	0.03±3.9%	373.16±2.7%	402.53±2.8%	409.50±3.0%	606.70±2.3%	19	23
Knuth-Yao Dice[2]	23.16±0.6%	0.60±8.3%	0.39±2.6%	421.52±3.9%	443.06±2.4%	436.38±2.2%	679.95±0.7%	74	83
Dining Phils. [36, B.1][2]	22.14±0.3%	0.83±4.8%	107.95±0.7%	551.27±3.2%	542.88±2.9%	931.20±1.9%	1262.56±0.6%	19579	58308
Synch. Leader Election[2]	22.58±0.5%	0.41±4.9%	0.02±4.3%	374.92±2.5%	404.71±2.7%	407.08±2.8%	619.37±0.7%	14	17
Monty Hall, Change	23.06±0.3%	0.43±4.7%	0.03±2.4%	380.95±2.7%	413.72±2.5%	418.95±2.1%	623.16±0.7%	20	23
Monty Hall, Stay	23.31±1.1%	0.44±4.5%	0.03±1.1%	380.74±2.4%	411.81±2.7%	412.80±2.7%	627.97±1.1%	20	23
Multiparty Workers[1]	20.63±1.6%	0.53±5.7%	3.26±0.3%	408.86±3.9%	415.49±3.1%	424.30±3.5%	638.86±0.9%	1222	3643
Gambler's Ruin	23.30±0.3%	0.42±4.8%	0.02±3.3%	376.44±2.0%	407.10±2.2%	418.77±2.5%	612.53±1.7%	16	19
Recursive Map-Reduce[1]	23.40±0.9%	0.46±6.5%	0.08±1.3%	379.64±2.9%	401.59±2.8%	400.31±2.5%	625.37±0.8%	68	119
Recursive Two Buyers[1]	24.82±4.6%	0.42±2.4%	0.02±3.0%	375.84±2.1%	406.04±2.4%	414.30±2.2%	607.63±1.1%	15	17
Example 4.16	24.51±6.1%	0.57±7.0%	0.05±1.7%	409.89±4.0%	414.10±1.8%	421.18±2.1%	625.69±0.8%	43	51
Von-Neumann Coin[2]	22.57±0.4%	0.39±2.6%	0.02±3.8%	371.26±2.0%	414.59±1.8%	412.39±2.2%	627.61±0.3%	16	19

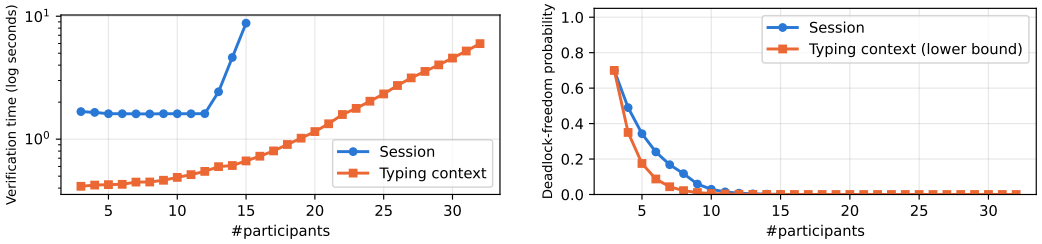
[1] Probabilistic variant of an example from [52].
[2] Adapted from the PRISM case studies [56].

standard error. For each benchmark process, we record the type-inference time and use the inferred typing context for subsequent measurements: translation into a PRISM model, precomputation of the weak almost-sure livelocked region (Def. 6.4), and separate PRISM invocations for the three properties of §6.1 (safety, deadlock-freedom, and liveness). We also measure the end-to-end runtime, including inference and translation, with a single PRISM invocation checking all three properties. As the translation function executes very quickly, each of its 30 data points is the mean of 100 consecutive calls (totalling 3,000 function calls), to reduce the effect of measurement overhead.

Table 1 presents the results, along with the size (number of states and transitions) of each context MDP. The benchmark processes and their type specifications are provided in the `/examples/` folder of the accompanying artifact [35]; measurements use freshly inferred contexts. The results of the verification process can be found in [36, App. E.5]. Most contexts are verified end-to-end in around 0.6*s*, and Dining Phils. (DP) in around 1.3*s*, demonstrating that Prose is suitable for practical use. Translation cost is negligible (<1ms), and the livelock-region precomputation is cheap (<6ms) on all contexts but DP, as the algorithm enumerates its large context state space. This makes PRISM model-checking the dominating cost. Model checking the same PRISM encoding with Storm [58] yielded similar results. End-to-end runtimes are close to those for individual properties, as PRISM's internal model construction, performed once per invocation, accounts for most of each runtime.

Comparison with model checking processes. To compare scalability of verifying session types and processes, we have measured verification times and deadlock-freedom probabilities of a simple factorial multiparty session M_i and its typing context Δ_i (for each $1 \leq i \leq 30$). While verifying sessions is undecidable in general (Proposition 2.3), this particular session can be translated into a finite-state PRISM module with one variable keeping track of its current state and one integer variable per received variable and per sent payload. Fig. 13a shows for $n \geq 14$, the PRISM invocations for session verification exhausted the allocated Java heap space (the default allocation is 1 GB), and consequently did not finish execution.

Moreover, Fig. 13b shows that the deadlock-freedom probability obtained from Δ_n is a lower bound of the precise deadlock-freedom probability of M_n, with both tending to 0 as n grows. As we describe in Example 4.19, the loss of precision comes from the conditional expressions in our calculus, which are typed as nondeterministic choice. See [36, App. E.6] for a more detailed explanation, as well as the factorial processes and types.



(a) Verification times (b) Deadlock-freedom probabilities

Fig. 13. Verification times and deadlock-freedom probabilities of the factorial session M_n which probabilistically produces an error; and its typing context Δ_n, with varying values of n. The verification times are a mean of 10 consecutive runs. For $n \geq 14$, the PRISM invocation for session M_n ran out of memory.

6.3 Prototype implementation of Type Inference & Checking

We implement a prototype of the type system, with roughly 1,500 lines of Haskell (GHC 9.6.7) (*type inference* and *subtype check* in Fig. 1). The tool takes as input a program: a list of named PROMT processes in a plain-text surface syntax mirroring §2, each optionally paired with a specified local type T_{spec}. The main challenge to inference is recursion: [T-REC] assumes the very type being inferred, while deriving a type for *conditionals* and *flips* requires knowledge of the type constructor of their branch types T_1 and T_2 (*conditionals* must compute the least upper bound of T_1 and T_2 with respect to $\leqslant$, and *flips* the merge $\mathsf{T}_1 \sqcap_p \mathsf{T}_2$).

Consider $P = \mu X.\mathbf{q}!\mathsf{m}.F$ where $F = \mathsf{flip}(p)\{\text{H}\Rightarrow X \parallel \text{T}\Rightarrow \mathbf{r}!\mathsf{m}.\mathbf{0}\}$. To type F requires the type of X, but finding that type requires typing F. We handle this circularity by recording type constraints and solving them on a finite graph. For example let $\mathsf{T}^?_X$ denote the *unknown* type of P (and X), and $\mathsf{T}^?_F$ the *unknown* type of F. Using the same type unknown $\mathsf{T}^?_X$ for the recursive process and its bound occurrences of X, the prototype seeks a solution of the equations: $\mathsf{T}^?_X = \oplus\{1 : \mathbf{q}!\mathsf{m}.\mathsf{T}^?_F\}$ and $\mathsf{T}^?_F = \mathsf{T}^?_X \sqcap_p \oplus\{1 : \mathbf{r}!\mathsf{m}.\mathsf{end}\}$.

The first equation determines an output node for $\mathsf{T}^?_X$ with an edge to the node reserved for $\mathsf{T}^?_F$, whose outgoing edges are still unknown. To infer them, the solver follows X to its binder and exposes the leading $\mathbf{q}!m$, retaining the reference to $\mathsf{T}^?_F$ beneath that send, yielding: $\mathsf{T}^?_F = \oplus\{1 : \mathbf{q}!\mathsf{m}.\mathsf{T}^?_F\} \sqcap_p \oplus\{1 : \mathbf{r}!\mathsf{m}.\mathsf{end}\}$. As both constructors under $\sqcap_p$ are revealed, we can merge to get: $\mathsf{T}^?_F = \oplus\{p : \mathbf{q}!\mathsf{m}.\mathsf{T}^?_F,\ (1-p) : \mathbf{r}!\mathsf{m}.\mathsf{end}\}$. The $\mathbf{q}!m$ continuation reuses the recorded node for F instead of unfolding the definition again, closing the cycle. Reading this graph as a recursive type yields $\mathsf{T}_{\mathsf{inf}} = \oplus\{1 : \mathbf{q}!\mathsf{m}.\mu\mathbf{t}.\ \oplus\{p : \mathbf{q}!\mathsf{m}.\mathbf{t},\ (1-p) : \mathbf{r}!\mathsf{m}.\mathsf{end}\}\}$.

The tool outputs, per process, the inferred local type $\mathsf{T}_{\mathsf{inf}}$, and optionally for each type specification a verdict of the subtype check $\mathsf{T}_{\mathsf{inf}} \leqslant \mathsf{T}_{\text{spec}}$, displaying both types on failure; Fig. 1 highlights the two workflows. Inferred types can also be exported to the input format of PROSE, our types to model checker translation §5, connecting inference directly to the verification pipeline.

The graph construction and the Cartesian products in [M-Σ] can both incur exponential growth, in the number of graph nodes and output summands respectively. This is consistent with the $O(n2^n)$ worst-case behaviour reported for bottom-up type inference in [59, Thm. 5.26], where $n = |P|$; the left-most column of Table 1 reports measured inference times for our benchmark programs.

7 Related Work and Conclusion

Probabilistic linear and binary session types. While many probabilistic concurrent models have been studied in process algebra (e.g., CCS [24], CSP [21, 45] and mobile process calculi [27, 48]),

few works use *behavioural types* to control and analyse probabilistic communications. The work in [60] applies linear types developed in [8, 63] to give a sound and complete correspondence between probabilistic processes and confusion-free event structures. Dal Lago and Giusti [40] develop a probabilistic binary (2-party) session calculus based on bounded linear logic, where typed processes execute within probabilistic polynomial time, enabling reasoning about cryptographic protocols.

Inverso et al. [34] propose a probabilistic type system which statically estimates an upper bound on the probability of typed processes reaching successful states, using the top-level probabilistic choice $P \boxplus_p Q$ for processes and types. Dal Lago and Padovani [41] study almost-sure termination of a binary session calculus with the same process-level probabilistic choice; their types carry no probability annotations, but real-valued *heights* bounding the expected number of reduction steps. This property is qualitative, whereas we derive quantitative bounds. Both calculi [34, 41] are built on the linear logic-based binary session type systems [13], expressing only tree-like network topologies and enforcing *total* (100%) deadlock-freedom, and every choice is probabilistic: there is no nondeterministic choice corresponding to our sums. Neither work provides an implementation.

Das et al. [17] propose probabilistic resource-aware session types extended from intuitionistic linear logic-based deadlock-free session types [13], implemented as the programming language NomosPro, which verifies representative probabilistic distributed algorithms with *exact probability* and *resource bounds*. Like our work, they provide both nondeterministic and probabilistic choice, and our flip construct and merging operator for types are inspired by NomosPro.

Our merge $\mathsf{T}_1 \sqcap_p \mathsf{T}_2$ (Def. 3.8) takes the convex combination of the outermost probabilities and takes the least upper bounds under $\leqslant$ for continuations that have a common prefix. Where the weighted sums of [17, 20] require these continuations to coincide, our merge accepts differing continuations whenever their join exists. E.g., consider a server that is equally likely to be healthy or degraded, acknowledging a request with probability 0.9 or 0.5, and then signalling hi or lo:

$$\mathsf{T}_H = \oplus\{\, 0.9 : \mathbf{q}!\mathsf{ok}.\oplus \mathbf{q}!\mathsf{hi}.\,\mathsf{end},\;\; 0.1 : \mathbf{q}!\mathsf{err}.\,\mathsf{end}\,\},\;\; \mathsf{T}_T = \oplus\{\, 0.5 : \mathbf{q}!\mathsf{ok}.\oplus \mathbf{q}!\mathsf{lo}.\,\mathsf{end},\;\; 0.5 : \mathbf{q}!\mathsf{err}.\,\mathsf{end}\,\}.$$

Then we have: $\mathsf{T}_H \sqcap_{0.5} \mathsf{T}_T = \oplus\{0.7 : \mathbf{q}!\mathsf{ok}.\,\Sigma\{\oplus \mathbf{q}!\mathsf{hi}.\,\mathsf{end},\; \oplus \mathbf{q}!\mathsf{lo}.\,\mathsf{end}\},\; 0.3 : \mathbf{q}!\mathsf{err}.\,\mathsf{end}\}$, while in [17, 20], this is undefined. The server remains typable despite its differing continuations: the probability of ok is retained exactly, while the subsequent choice between hi and lo is abstracted by nondeterminism. The work in [17] is extended to probabilistic refinement session types [20], augmented by symbolic probabilities indexed by refinement variables, enabling parameterised probabilistic reasoning and analyses of expected resource costs. Their implementation reconstructs the intermediate types of channels with an SMT solver, verifying *declared* types, whereas type-checking in our system is decidable (Theorem 3.14). Their implementation of Lehmann and Rabin's randomised dining philosophers [43] uses a coordinating process and an indexed ring, with refinement predicates tracking fork availability and ownership. Our multiparty formulation ([36, Ex. B.1]) expresses contention directly through separated external choice: each fork accepts requests from either neighbour, then awaits its holder's release while rejecting competing requests. Ownership is captured by the session-type continuations, without refinement indices or a central process mediating fork acquisition.

None of the above calculi study probabilistic deadlock-freedom or liveness; processes with only partial deadlock-freedom are untypable.

Probabilistic multiparty session types (MPST) and choreographies. Aman and Ciobanu [1, 2] extend MPST [30, 31] with probability intervals on selections in global and local types. Their top-down approach checks conformance to interval-annotated global protocols. Our bottom-up approach [59] instead analyses locally typed sessions and establishes lower bounds on their probabilities of deadlock-freedom and liveness, including sessions for which these properties do not hold

on every fair execution. Blechschmidt et al. [10] propose a probabilistic mixed choice multiparty session π-calculus. Their refinement subtyping permits replacement of a channel by several channels implementing the same behaviour, differing from the standard subtyping relation defined between two local types. The works in [1, 2, 10] provide no implementation. Carbone and Veschetti [14] propose a probabilistic deadlock-free choreographic calculus with choices explicitly annotated by probabilities (akin to global types in [1, 2]), prove the correctness of its translation into PRISM, and compare outcomes of probabilistic measurements against the original PRISM case studies. Their language more closely resembles the PRISM language than our local types: a choreographic sending-receiving construct describes both endpoints of a communication at once, mapping directly to one action in PRISM, whereas we translate each participant of a typing context into its own PRISM module (§5.2.2), with a communication arising as a synchronisation between the sender's and receiver's modules. Proving that each reduction of the typing context corresponds to exactly one transition of the composed model (Thms. 5.6 and 5.7) requires a closure module blocking spurious unsynchronised transitions and a bisimulation relating models under recursion unfolding, making our correctness proofs much more involved. No works above analyse probabilistic bounds for properties such as deadlock-freedom and liveness.

Model checking behavioural types and programs. We translate *types* rather than processes to the PRISM language because: (1) verification of session types is guaranteed to terminate, whereas even safety of processes is undecidable (Proposition 2.3); and (2) translating processes results in a far larger state space, as exact values of every variable must be tracked, making verification of types more efficient and scalable, as shown in Fig. 13 and [36, App. E.6].

Model checking real-world programs has a long history [61]. Chaki et al. [16] are the first to propose model checking behavioural types (an abstraction of π-processes) developed in [33]. More recently, [52] applied mCRL2 [12] to verify multiparty session types, an approach subsequently extended to Go [42] and Scala [53] programs, and to distributed failures [7]. None of the above are probabilistic.

Beyond behavioural types, Enea et al. [19] verify *fair almost-sure termination* of randomized distributed algorithms such as Ben-Or's consensus, using a proof rule combining martingale-based termination arguments with weak fairness, discharged by an SMT solver. Their deductive approach works directly on processes whose state spaces may grow unboundedly, beyond the reach of finite-state model checking; conversely, they establish termination with probability one, whereas we derive quantitative lower bounds on deadlock-freedom and liveness.

Conclusion. We have presented the first probabilistic multiparty session type theory for deriving probabilistic properties (probabilistic deadlock-freedom and liveness) of typed multiparty session processes. Our nondeterministic sums abstract data-dependent control, inducing lower bounds on probabilistic properties that hold under every fair adversarial scheduler. We have implemented the tool-chain Prose based on the precise translation from session types to PRISM, and evaluated it on several case studies. Our future work includes the extension of the calculus with session delegations [31, 52], dynamically joining participants [15, 18] and asynchronous sessions [51]. Our subtyping system is *not* precise in the sense of [22, 23, 46]. For example, $\oplus\{0.5 : \mathbf{p}!\mathsf{m}, 0.5 : \mathbf{q}!\mathsf{m}\} \not\leq \Sigma\{\oplus\{\mathbf{p}!\mathsf{m}\}, \oplus\{\mathbf{q}!\mathsf{m}\}\}$, although the distribution on the left is a convex combination of the alternatives on the right. Investigating probabilistic bisimulations [54] to refine subtyping and minimise session types is another direction for future work. We also plan to analyse more advanced properties such as real-time session type specifications [32, 62], taking advantage of PRISM's support for probabilistic timed automata.

Acknowledgements. We thank the POPL reviewers for their careful comments, encouragement and detailed suggestions. We thank Jake Masters for discussions and Kai Pischke for testing our artifact.

This work is supported by ARIA MSAI-PR01-P020 and MSAI-PR01-P029; and EPSRC EP/T006544/2, EP/T014709/2, EP/Z533749/1 and EP/Y005244/1; Horizon EU TaRDIS 101093006 (UKRI number 10066667); and a grant from the Simons Foundation.

8 Data Availability Statement

The implementation of Prose, as well as scripts and instructions to reproduce the experimental results of this paper as well as the prototype for type-checking and type-inference are included as an artifact available on Zenodo [35].

9 Disclosure of Use of AI in Research

We used OpenAI's GPT-6 Astra [49], accessed through Codex, to assist with the implementation of the type-inference prototype promt, and to generate regression tests and explore counterexamples. We additionally used Anthropic's Claude Fable [4], accessed through Claude Code, to assist with the implementation of Prose and to generate benchmarking scripts. The authors directed this work and take responsibility for the implementation and reported results.

References

[1] Bogdan Aman and Gabriel Ciobanu. 2019. Probabilities in Session Types. *Electronic Proceedings in Theoretical Computer Science* 303 (Sept. 2019), 92–106. doi:10.4204/eptcs.303.7

[2] Bogdan Aman and Gabriel Ciobanu. 2022. Interval Probability for Sessions Types. In *Logic, Language, Information, and Computation - 28th International Workshop, WoLLIC 2022, Iaşi, Romania, September 20-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13468)*, Agata Ciabattoni, Elaine Pimentel, and Ruy J. G. B. de Queiroz (Eds.). Springer, 123–140. doi:10.1007/978-3-031-15298-6_8

[3] Roman Andriushchenko, Alexander Bork, Carlos E. Budde, Milan Češka, Kush Grover, Ernst Moritz Hahn, Arnd Hartmanns, Bryant Israelsen, Nils Jansen, Joshua Jeppson, Sebastian Junges, Maximilian A. Köhl, Bettina Könighofer, Jan Křetínský, Tobias Meggendorfer, David Parker, Stefan Pranger, Tim Quatmann, Enno Ruijters, Landon Taylor, Matthias Volk, Maximilian Weininger, and Zhen Zhang. 2024. Tools at the Frontiers of Quantitative Verification: QComp 2023 Competition Report. In *TOOLympics Challenge 2023: Updates, Results, Successes of the Formal-Methods Competitions (Lecture Notes in Computer Science, Vol. 14550)*, Dirk Beyer, Arnd Hartmanns, and Fabrice Kordon (Eds.). Springer, Cham, 90–146. doi:10.1007/978-3-031-67695-6_4

[4] Anthropic. 2026. Claude Fable 5.1. https://platform.claude.com/docs/en/models/fable-5-1/overview Model documentation. Accessed 24 October 2026.

[5] C. Baier. 1998. On Algorithmic Verification Methods for Probabilistic Systems. (1998). Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim.

[6] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking (Representation and Mind Series).* The MIT Press.

[7] Adam Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. 2022. Generalised Multiparty Session Types with Crash-Stop Failures. In *33rd International Conference on Concurrency Theory (LIPIcs, Vol. 243)*. Dagstuhl, 35:1–35:25. doi:10.4230/LIPIcs.CONCUR.2022.35

[8] Martin Berger, Kohei Honda, and Nobuko Yoshida. 2001. Sequentiality and the π-Calculus. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science, Vol. 2044)*, Samson Abramsky (Ed.). Springer, Berlin, Heidelberg, 29–45. doi:10.1007/3-540-45413-6_7

[9] Andrea Bianco and Luca de Alfaro. 1995. Model Checking of Probabilistic and Nondeterministic Systems. In *Foundations of Software Technology and Theoretical Computer Science (Lecture Notes in Computer Science, Vol. 1026)*, P. S. Thiagarajan (Ed.). Springer, Berlin, Heidelberg, 499–513. doi:10.1007/3-540-60692-0_70

[10] Paula Blechschmidt, Kirstin Peters, and Uwe Nestmann. 2025. Compositional Interface Refinement Through Subtyping in Probabilistic Session Types. In *Theoretical Aspects of Computing – ICTAC 2025: 22nd International Colloquium, Marrakech, Morocco, November 24–28, 2025, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 145–163. doi:10.1007/978-3-032-11176-0_10

[11] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. 2017. JANI: Quantitative Model and Tool Interaction. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science, Vol. 10206)*, Axel Legay and Tiziana Margaria (Eds.). Springer, Berlin, Heidelberg, 151–168. doi:10.1007/978-3-662-54580-5_9

[12] 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. 2019. The mCRL2 Toolset for Analysing Concurrent Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 21–39.

[13] Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Math. Struct. Comput. Sci.* 26, 3 (2016), 367–423. doi:10.1017/S0960129514000218

[14] Marco Carbone and Adele Veschetti. 2024. A Probabilistic Choreography Language for PRISM. In *Coordination Models and Languages (Lecture Notes in Computer Science, Vol. 14676)*, Ilaria Castellani and Francesco Tiezzi (Eds.). Springer, Cham, 20–37. doi:10.1007/978-3-031-62697-5_2

[15] David Castro-Perez and Nobuko Yoshida. 2023. Dynamically Updatable Multiparty Session Protocols: Generating Concurrent Go Code from Unbounded Protocols. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:30. doi:10.4230/LIPIcs.ECOOP.2023.6

[16] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. 2002. Types as Models: Model Checking Message-Passing Programs. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 45–57. doi:10.1145/503272.503278

[17] Ankush Das, Di Wang, and Jan Hoffmann. 2023. Probabilistic Resource-Aware Session Types. *Proc. ACM Program. Lang.* 7, POPL, Article 66 (Jan. 2023), 32 pages. doi:10.1145/3571259

[18] Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *Proceedings of POPL 2011*. ACM, 435–446. doi:10.1145/1926385.1926435

[19] Constantin Enea, Rupak Majumdar, Harshit Jitendra Motwani, and V. R. Sathiyanarayana. 2026. Verifying Almost-Sure Termination for Randomized Distributed Algorithms. *Proc. ACM Program. Lang.* 10, POPL, Article 49 (Jan. 2026), 30 pages. doi:10.1145/3776691

[20] Qiancheng Fu, Ankush Das, and Marco Gaboardi. 2025. Probabilistic Refinement Session Types. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1666–1691. doi:10.1145/3729317

[21] Sonja Georgievska and Suzana Andova. 2012. Probabilistic CSP: Preserving the Laws via Restricted Schedulers. In *Proceedings of the International GI/ITG Conference on Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance (MMB/DFT)*, Vol. 7201. Springer, 136–150. doi:10.1007/978-3-642-28540-0_10

[22] Silvia Ghilezan, Svetlana Jakšić, Jovanka Pantović, Alceste Scalas, and Nobuko Yoshida. 2019. Precise Subtyping for Synchronous Multiparty Sessions. *Journal of Logical and Algebraic Methods in Programming* 104 (April 2019), 127–173. doi:10.1016/j.jlamp.2018.12.002

[23] Silvia Ghilezan, Jovanka Pantović, Ivan Prokić, Alceste Scalas, and Nobuko Yoshida. 2023. Precise Subtyping for Asynchronous Multiparty Sessions. *ACM Trans. Comput. Logic* 24, 2, Article 14 (March 2023), 73 pages. doi:10.1145/3568422

[24] Hans A. Hansson. 1992. Time and probabilities in specification and verification of real-time systems. In *Proceedings of the Euromicro workshop on Real-Time Systems (RTS)*. 92–97. doi:10.1109/EMWRT.1992.637477

[25] Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 593–598. doi:10.1007/978-3-642-54862-8_51

[26] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The Probabilistic Model Checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (2022), 589–610. doi:10.1007/s10009-021-00633-z

[27] Oltea Mihaela Herescu and Catuscia Palamidessi. 2000. Probabilistic Asynchronous π-Calculus. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science, Vol. 1784)*, Jerzy Tiuryn (Ed.). Springer, Berlin, Heidelberg, 146–160. doi:10.1007/3-540-46432-8_10

[28] Ruth Hoffmann, Murray L. Ireland, Alice Miller, Gethin Norman, and Sandor M. Veres. 2016. Autonomous Agent Behaviour Modelled in PRISM - A Case Study. In *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9641)*, Dragan Bosnacki and Anton Wijs (Eds.). Springer, 104–110. doi:10.1007/978-3-319-32582-8_7

[29] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Programming Languages and Systems (Lecture Notes in Computer Science, Vol. 1381)*, Chris Hankin (Ed.). Springer, Berlin, Heidelberg, 122–138. doi:10.1007/BFb0053567

[30] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. Association for Computing Machinery, New York, NY, USA, 273–284. doi:10.1145/1328438.1328472

[31] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *Journal of the ACM* 63, 1 (2016), 9:1–9:67. doi:10.1145/2827695

[32] Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida. 2024. Fearless Asynchronous Communications with Timed Multiparty Session Protocols. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:30. doi:10.4230/LIPIcs.ECOOP.2024.19

[33] Atsushi Igarashi and Naoki Kobayashi. 2004. A Generic Type System for the π-Calculus. *Theoretical Computer Science* 311, 1–3 (2004), 121–163. doi:10.1016/S0304-3975(03)00325-6

[34] Omar Inverso, Hernán Melgratti, Luca Padovani, Catia Trubiani, and Emilio Tuosto. 2020. Probabilistic Analysis of Binary Sessions. In *31st International Conference on Concurrency Theory (CONCUR 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 171)*, Igor Konnov and Laura Kovács (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 14:1–14:21. doi:10.4230/LIPIcs.CONCUR.2020.14

[35] A. Junge, L. S. Lee, J. Paulus, N. Yoshida, and D. Parker. 2026. Artifact for "Model-Checking Probabilistic Multiparty Session Types". Zenodo. doi:10.5281/zenodo.23326139 Computer software.

[36] Aleksander Junge, Leo Seojun Lee, Joseph Paulus, Nobuko Yoshida, and David Parker. 2026. Model Checking Probabilistic Multiparty Session Types. arXiv:2611.14795 [cs.PL] doi:10.48550/arXiv.2611.14795

[37] John G. Kemeny, J. Laurie Snell, and Anthony W. Knapp. 1976. *Denumerable Markov Chains* (2 ed.). Graduate Texts in Mathematics, Vol. 40. Springer, New York. doi:10.1007/978-1-4684-9455-6 With a chapter on Markov random fields by David Griffeath.

[38] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, Berlin, Heidelberg, 585–591. doi:10.1007/978-3-642-22110-1_47

[39] Marta Kwiatkowska, Gethin Norman, and David Parker. 2025. Probabilistic Model Checking: Applications and Trends. In *Principles of Formal Quantitative Analysis: Essays Dedicated to Christel Baier on the Occasion of Her 60th Birthday*, Nathalie Bertrand, Clemens Dubslaff, and Sascha Klüppelholz (Eds.). Lecture Notes in Computer Science, Vol. 15760. Springer, Cham, 158–173. doi:10.1007/978-3-031-97439-7_7

[40] Ugo Dal Lago and Giulia Giusti. 2022. On Session Typing, Probabilistic Polynomial Time, and Cryptographic Experiments. In *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland (LIPIcs, Vol. 243)*, Bartek Klin, Slawomir Lasota, and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 37:1–37:18. doi:10.4230/LIPICS.CONCUR.2022.37

[41] Ugo Dal Lago and Luca Padovani. 2024. On the Almost-Sure Termination of Binary Sessions. In *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming, PPDP 2024, Milano, Italy, September 9-11, 2024*, Alessandro Bruni, Alberto Momigliano, Matteo Pradella, Matteo Rossi, and James Cheney (Eds.). ACM, 9:1–9:12. doi:10.1145/3678232.3678239

[42] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *Proceedings of ICSE 2018*. ACM, 1137–1148. doi:10.1145/3180155.3180157

[43] Daniel J. Lehmann and Michael O. Rabin. 1981. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. Association for Computing Machinery, Williamsburg, Virginia, USA, 133–138. doi:10.1145/567532.567547

[44] Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (Nov. 1994), 1811–1841. doi:10.1145/197320.197383

[45] Gavin Lowe. 1995. Probabilistic and Prioritized Models of Timed CSP. *Theor. Comput. Sci.* 138, 2 (1995), 315–352. doi:10.1016/0304-3975(94)00171-E

[46] Jake Masters and Nobuko Yoshida. 2027. Mixed Choice Multiparty Session Types, Precisely. In *Programming Languages and Systems (ESOP 2027)*. Copenhagen, Denmark. arXiv:2608.10704 [cs.LO] https://arxiv.org/abs/2608.10704 To appear. Full version available on arXiv.

[47] Marvin L. Minsky. 1967. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, N.J. xviii+317 pages.

[48] Gethin Norman, Catuscia Palamidessi, David Parker, and Peng Wu. 2007. Model Checking the Probabilistic π-Calculus. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*. IEEE Computer Society, Edinburgh, UK, 169–178. doi:10.1109/QEST.2007.31

[49] OpenAI. 2026. GPT-6 Astra. https://developers.openai.com/api/docs/models/gpt-6-astra Model documentation. Accessed 20 October 2026.

[50] Kirstin Peters and Nobuko Yoshida. 2024. Separation and Encodability in Mixed Choice Multiparty Sessions. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science* (Tallinn, Estonia) *(LICS '24)*. Association for Computing Machinery, New York, NY, USA, Article 62, 15 pages. doi:10.1145/3661814.3662085

[51] Ivan Prokić, Simona Prokić, Silvia Ghilezan, Alceste Scalas, and Nobuko Yoshida. 2026. On Asynchronous Multiparty Session Types for Federated Learning. *Theoretical Computer Science* (2026), 116286. doi:10.1016/j.tcs.2026.116286

[52] Alceste Scalas and Nobuko Yoshida. 2019. Less Is More: Multiparty Session Types Revisited. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–29. doi:10.1145/3290343

[53] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying Message-Passing Programs with Dependent Behavioural Types. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 502–516. doi:10.1145/3314221.3322484

[54] Roberto Segala and Nancy A. Lynch. 1995. Probabilistic Simulations for Probabilistic Processes. *Nord. J. Comput.* 2, 2 (1995), 250–273.

[55] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and Its Typing System. In *PARLE '94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings (Lecture Notes in Computer Science, Vol. 817)*, Constantine Halatsis, Dimitris G. Maritsas, George Philokyprou, and Sergios Theodoridis (Eds.). Springer, 398–413. doi:10.1007/3-540-58184-7_118

[56] PRISM Team. 2026. PRISM case studies. https://www.prismmodelchecker.org/casestudies/.

[57] PRISM Team. 2026. PRISM documentation. https://www.prismmodelchecker.org/doc/.

[58] Storm Team. 2025. Storm homepage (version 1.10.0). https://www.stormchecker.org/.

[59] Thien Udomsrirungruang and Nobuko Yoshida. 2025. Top-Down or Bottom-Up? Complexity Analyses of Synchronous Multiparty Session Types. *Proc. ACM Program. Lang.* 9, POPL (2025), 1040–1071. doi:10.1145/3704872

[60] Daniele Varacca and Nobuko Yoshida. 2007. Probabilistic π-calculus and event structures. *Electronic Notes in Theoretical Computer Science* 190, 3 (2007), 147–166. doi:10.1016/j.entcs.2007.07.009

[61] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232. doi:10.1023/A:1022920129859

[62] Yue Yao, Grant Iraci, Cheng-En Chuang, Stephanie Balzer, and Lukasz Ziarek. 2025. Semantic Logical Relations for Timed Message-Passing Protocols. *Proc. ACM Program. Lang.* 9, POPL (2025), 1750–1781. doi:10.1145/3704895

[63] Nobuko Yoshida, Martin Berger, and Kohei Honda. 2004. Strong normalisation in the pi -calculus. *Inf. Comput.* 191, 2 (2004), 145–202. doi:10.1016/j.ic.2003.08.004

[64] Nobuko Yoshida and Ping Hou. 2024. Less is More Revisited. In *The Practice of Formal Methods (Lecture Notes in Computer Science, Vol. 14781)*, Ana Cavalcanti and James Baxter (Eds.). Springer, Cham, 268–291. doi:10.1007/978-3-031-66673-5_14