Less Is More: Multiparty Session Types Revisited

(Technical Report)

ALCESTE SCALAS, Imperial College London, UK
NOBUKO YOSHIDA, Imperial College London, UK

Multiparty Session Types (MPST) are a typing discipline ensuring that a message-passing process implements a given multiparty session protocol, without errors. In this paper, we propose a new, generalised MPST theory.

Our contribution is fourfold. (1) We demonstrate that a revision of the theoretical foundations of MPST is necessary: classic MPST have a limited subject reduction property, with inherent restrictions that are easily overlooked, and in previous work have led to flawed type safety proofs: our new theory removes such restrictions and fixes such flaws. (2) We contribute a new MPST theory that is less complicated, and yet more general, than the classic one: it does not require global multiparty session types nor binary session type duality — instead, it is grounded on general behavioural type-level properties, and proves type safety of many more protocols and processes. (3) We produce a detailed analysis of type-level properties, showing how, in our new theory, they allow to ensure decidability of type checking, and statically guarantee that processes enjoy, e.g., deadlock-freedom and liveness at run-time. (4) We show how our new theory can integrate type and model checking: type-level properties can be expressed in modal μ-calculus, and verified with well-established tools.

CCS Concepts: • Theory of computation → Process calculi; Type structures; Verification by model checking;

Additional Key Words and Phrases: session types, duality, deadlock-freedom, liveness

1 INTRODUCTION

Session types are a type-based framework for formalising structured communication protocols, and verifying them in concurrent message-passing programs. The original binary session types theory [Honda et al. 1998] addresses protocols with two participants (e.g., client and server), and is built on a notion of duality in interactions, inspired by linear logic [Girard 1987]; this has led to several studies on the logical foundations for session types, e.g. [Caires et al. 2016; Wadler 2014]. This approach was later generalised to multiparty sessions [Bettini et al. 2008; Honda et al. 2008], supporting more sophisticated protocols with any number of participants (two or more); correspondingly, binary duality was generalised as multiparty consistency, leading to studies on its logical foundations [Caires and Pérez 2016; Carbone et al. 2016, 2015].

Unfortunately, this duality-based framework has intrinsic limitations: the consistency requirement is not satisfied by many multiparty protocols — even surprisingly simple ones. Such limitations are subtle: in this paper, we show that they have been overlooked or wrongly bypassed in several previous works, leading to MPST extensions that are no longer correct, and have flawed subject reduction proofs. Then, we provide a solution: a new, generalised MPST theory that subsumes classic MPST under a new theoretical foundation, removes its limitations, fixes the aforementioned flaws, and supports a richer set of multiparty protocols and processes.

The Multiparty Session Types (MPST) framework. Bettini et al. [2008]; Honda et al. [2008] introduce the seminal notion of global types, which describe multiparty conversations from a global perspective. MPST verification follows a top-down approach based on endpoint projections.

This is the long version of the POPL’19 paper with the same title, and by the same authors: https://doi.org/10.1145/3290343

Authors’ addresses: Alceste Scalas, Imperial College London, UK, alceste.scalas@imperial.ac.uk; Nobuko Yoshida, Imperial College London, UK, n.yoshida@imperial.ac.uk.
(1) a multiparty protocol is formalised as a global type \( G \), providing a bird’s eye view on the interactions between two or more roles; (2) \( G \) is projected onto a set of endpoint (local) session types (one per role); and (3) session types are assigned to communication channels, used by MPST processes that can be written and type-checked separately.

E.g., the global type \( G \) below models a protocol (based on OAuth 2.0 [OAuth Working Group 2012]) between service \( s \), client \( c \), and authorisation server \( a \):

\[
G = s \rightarrow c: \{ \text{login}.c \rightarrow a:\text{passwd(Str)}.a \rightarrow s:\text{auth}(Bool).\text{end}, \text{cancel}.c \rightarrow a:\text{quit}.\text{end} \}
\]

The protocol of \( G \) says that the service sends to the client either a request to login, or cancel; in the first case, \( c \) continues by sending passwd (carrying a String) to the authorisation server, who in turn sends auth to \( s \) (with a Boolean, telling whether the client is authorised), and the session ends; in the second case, \( c \) sends quit to \( a \), and the session ends. The projections of \( G \) describe the local I/O actions (i.e., the interfaces) that programs must implement to play the roles in \( G \):

\[
\begin{align*}
S_s &= c@\{ \text{login}.a@\text{auth}(Bool), \text{cancel} \} \\
S_c &= s& \{ \text{login}.a@\text{passwd(Str)}, \text{cancel}.a@\text{quit} \} \\
S_a &= c& \{ \text{passwd(Str)}.s@\text{auth}(Bool), \text{quit} \}
\end{align*}
\]

Here, \( S_s, S_c, S_a \) are session types, obtained by projecting \( G \) resp. onto \( s, c, a \) (for brevity, we omit final ends). \( S_s \) represents the interface of \( S \) in \( G \): it must send (\( @ \)) to \( c \) either login or cancel; in the first case, \( s \) must then receive (\( & \)) message auth(Bool) from \( a \), and the session ends; otherwise, in the second case, the session just ends. Types \( S_c \) and \( S_a \) follow the same intuition. The multiparty session type system assigns the types in (2) to channels, and checks that endpoint programs use them correctly: e.g., the program implementing the service is checked against \( S_s \), and the programs implementing \( c/a \) against \( S_c/S_a \). Endpoint programs, in turn, are formalised as processes in a \( \pi \)-calculus extended with multiparty communication primitives. Variations of this framework have been implemented in numerous programming languages (surveyed in Ancona et al. [2017]; Gay and Ravara [2017]), allowing to develop distributed applications with guaranteed protocol conformance.

**Limitations and Theoretical Issues of MPST.** Theories and implementations based on MPST crucially require “correct by construction” protocols that do not cause deadlocks nor communication errors when endpoint programs interact. This is achieved by imposing well-formedness conditions to global types, and consistency restrictions when processes are type-checked.

However, such restrictions introduce rather serious problems when proving subject reduction — i.e., when proving that typed processes only reduce to typed processes, and thus, no (untypable) error state can be reached (“typed processes never go wrong”). Usually, one expects a statement like:

\[
\Gamma \vdash P \quad \text{and} \quad P \rightarrow P' \quad \text{implies} \quad \exists \Gamma' : \Gamma' \vdash P'
\]

where \( \Gamma \vdash P \) is a typing judgement stating that process \( P \) abides by the typing context \( \Gamma \), which can map, e.g., the communication channels \( c_s, c_c, c_a \) to the types \( S_s, S_c, S_a \) in (2).

Unfortunately, (3) is wrong. If we take \( \Gamma \) without any constraint as in (3), it might contain types like \( c@m(\text{Str}).\text{end} \) and \( s&m(\text{Int}).\text{end} \), and they could type a parallel process \( P = P_1 \upharpoonright P_2 \), where \( P_1 \) and \( P_2 \) interact according to the types, with \( P_1 \) sending a message \( m(“Hello”) \) (carrying a String), and \( P_2 \) receiving \( m \) but using its payload as an Integer. In this case, \( P \) would reduce to a “wrong” and untypable \( P' \) (see also [Coppe et al. 2015a, p. 163], and §3 later on): this means that (3) does not hold. For this reason, the MPST theory requires the aforementioned consistency restriction, and its actual subject reduction statement reads:

\[
\Gamma \vdash P \quad \text{with} \quad \Gamma \text{ consistent} \quad \text{and} \quad P \rightarrow P' \quad \text{implies} \quad \exists \Gamma' \text{ consistent}: \Gamma' \vdash \Gamma' \text{ and } \Gamma' \vdash P'
\]

Alceste Scalas and Nobuko Yoshida
(where $\Gamma \rightarrow \Gamma'$ denotes typing context reductions). Consistency is a syntactic constraint ensuring that the potential output messages of each role match the input capabilities of their recipient; as noted above, this requirement was developed by generalising the notion of binary session duality [Honda et al. 1998]. However, due to this binary session heritage, multiparty consistency is:

1) **overly restrictive.** Consistency does not hold for many protocols: even the simple authorisation protocol in (1)/(2) above is not consistent. Hence, for such protocols, the MPST framework cannot prove type safety of any process, because (4) holds vacuously;

2) **inflexible and error-prone.** Some MPST works, e.g. [Deniélou et al. 2012; Deniélou and Yoshida 2012; Yoshida et al. 2010], propose richer global types with flexible well-formedness conditions — but either overlook the consistency requirement, or fail to realise that their extensions do not satisfy it. Hence, their subject reduction theorems do not hold (like (3)), or hold vacuously (as above); and worryingly, such results are reused in later works and implementations (more details in §8).

These two claims are based on technical arguments, that we develop in §3. They clearly undermine the expressiveness and applicability of MPST: when the theory cannot ensure type safety for a given protocol, MPST-based implementations should either reject it (thus being overly restrictive), or forfeit the guaranteed absence of run-time errors. To solve these problems, we pose the questions:

- *Can we remove the duality/consistency requirements of MPST?*
- *Can we use, instead, more flexible properties of session types, thus enlarging the subject reduction property, and the set of provably type-safe processes?*

To answer positively, we need a new MPST theory that is not rooted in binary session duality — but has more general foundations, that still support duality as a special case.

**Contributions.** We present a new theory of multiparty session types. Its novel theoretical foundations leverage a weak behavioural safety invariant that, for the first time, eschews the limitations of duality/consistency, and allows to obtain much more general results than classic MPST.

We summarise MPST definitions and typing rules in §2, highlighting where our new theory diverges from the classic (§2.3): i.e., when establishing the prerequisites for proving type safety.

1) *We explain how classic MPST establish such prerequisites: i.e., by imposing consistency/duality. We uncover that the resulting severe limitations lead to subtle theoretical issues (§3).*

2) *We present our new MPST theory (§4), with a much weaker prerequisite: a safety invariant, not depending on global types, nor needing projection/duality/consistency from classic MPST.*

3) *By removing consistency, we rebuild the theoretical foundations of MPST on a more general basis. Our rebuilding subsumes classic MPST works, and fixes their theoretical issues, by producing more general typing rules, with just small visible differences (Remark 5.12).*

4) *We design our new type system to be parametric: its safety invariant is abstracted as a parameter $\phi$. We show that $\phi$ can be fine-tuned to ensure decidability of type-checking, and statically enforce various run-time properties on processes — e.g., liveness (§5.3, §5.4, §5.5).*

5) *The parameter $\phi$ can be a behavioural property: this allows for a novel integration of type/model checking techniques for MPST. We show how to express $\phi$ as a modal $\mu$-calculus formula, and verify type-level properties via model checking, using the paper’s companion artifact (§6). Via point 4 above, the model-checked properties transfer to processes.*

6) *Our theory extends to asynchronous communication, to handle richer protocols and programs. Asynchrony makes $\phi$ (and type checking) undecidable; still, we present various ways to achieve decidable type checking, with methods based e.g. on communicating automata (§7).*

**NOTE:** The main difference between this technical report the conference paper [Scalas and Yoshida 2019] are the appendices, that contain technical details, proofs, and discussion on related work: we
provide some pointers in the main text. In particular, in §H, we discuss more related topics (e.g., asynchronous subtyping), and other theories that have different goals, or cannot handle our examples (conversation types, choreographic programming).

2 MULTIPARTY SESSION TYPES

This section describes the multiparty session π-calculus (§2.1), its types, and typing rules (§2.2). Our streamlined formulation is based on Coppo et al. [2015a] and Scalas et al. [2017a], i.e., the most common in literature; we include subtyping [Dezani-Ciancaglini et al. 2015], to later study its crucial influence on the behavioural properties of types and processes (§5).

Crucially, in this section we leave one typing rule under-specified: the rule for session restriction. The reason is explained in §2.3: the exact form of this rule strictly depends on the theoretical foundations that allow to prove type safety — and the choice of such foundations is the crossroads where our new theory (§4) departs from classic MPST (§3).

2.1 The Multiparty Session π-Calculus

The multiparty session π-calculus models processes that interact via multiparty channels. We give a streamlined definition, sufficient for our developments. Extensions with, e.g., ground values (booleans, strings,…), or conditionals, are standard and orthogonal; we use them in examples.

Definition 2.1. The multiparty session π-calculus syntax is defined as follows:

\[
c, d ::= x \mid s[p] \quad \text{(variable, channel with role p)}
\]

\[
P, Q ::= 0 \mid P \mid Q \mid (\nu s) P \quad \text{(inaction, composition, restriction)}
\]

\[
c[q]@m(d).P \quad \text{(selection towards role q)}
\]

\[
c[q]\sum_{i \in I} m_i(x_i).P_i \quad \text{(branching from role q with } I \neq 0\text{)}
\]

\[
def D \in P \mid X(\overline{c}) \mid \text{err} \quad \text{(process definition, process call, error)}
\]

\[
D ::= X(\overline{x}) = P \quad \text{(declaration of process variable X)}
\]

Restriction, branching and declarations act as binders, as expected; \(\text{fc}(P)\) is the set of free channels with roles in \(P\), and \(\text{fv}(P)\) is the set of free variables in \(P\). We adopt a form of Barendregt convention: bound sessions and variables are assumed pairwise distinct, and different from free ones.

A channel \(c\) can be either a variable or a channel with role \(s[p]\), i.e., a multiparty communication endpoint whose user plays role \(p\) in the session \(s\). The inaction \(0\) represents a terminated process (and is often omitted). The parallel composition \(P \mid Q\) represents two processes that can execute concurrently, and potentially communicate. The session restriction \((\nu s) P\) declares a new session \(s\) with scope limited to process \(P\). Process \(c[q]@m(d).P\) performs a selection (internal choice) towards role \(q\), using the channel \(c\): the message label \(m\) is sent with the payload \(d\), and the execution continues as \(P\). Dually, the branching (external choice) \(c[q]\sum_{i \in I} m_i(x_i).P_i\) uses channels \(c\) to wait for a message from role \(q\): if a message label \(m_k\) with payload \(d\) is received (for some \(k \in I\)), then the execution continues as \(P_k\), with \(x_k\) replaced by \(d\). Note that variable \(x_i\) is bound with scope \(P_i\). Process definition \(D = X(\overline{x}) \in P\) in \(Q\) and process call \(X(\overline{c})\) model recursion: the call invokes \(X\) by expanding it into \(P\), and replacing its formal parameters with the actual ones. \(\text{err}\) denotes the error process. Note that our simplified syntax does not have “pure” input/output prefixes: they can be easily encoded as singleton branch/selection.

Definition 2.2 (Semantics). A reduction context \(\mathcal{C}\) is:

\[
\mathcal{C} ::= \mathcal{C} \mid P \mid (\nu s)\mathcal{C} \mid \text{def } D \in \mathcal{C} \mid []
\]

\(\text{Reduction} \rightarrow\) is inductively defined in Fig. 1, up-to a standard structural congruence \(\equiv\) (§A) including \(\alpha\)-conversion. We say that \(P\) has an error iff, for some \(\mathcal{C}\), \(P = \mathcal{C}[\text{err}]\).
In Def. 2.2, the reduction context $C$ defines a process with a single hole [], occurring in place of some subterm $P$. The communication rule [R-Comm] says that the parallel composition of a branching and a selection process, both operating on the same session $s$ respectively as roles $p$ and $q$, reduces to the corresponding continuations, with the sent channel being substituted on the receiver side. The process call rule [R-X] allows to invoke the process $P$ in the definition of $X$ by creating a copy of $P$, and replacing the formal parameters $x_i$ with actual parameters, i.e., channels with role $s_i[p_i]$. The standard context rule [R-Ctx] says that reduction can happen under parallel composition, restriction and process definition (cf. definition of $C$). Finally, the error rule [R-Err] says that a parallel composition of mismatching selection and branching processes reduces to err: intuitively, it models a scenario where a process implementing role $q$ is trying to send $m$ to another process implementing $p$ — who is indeed waiting for an input, but does not expect to receive $m$.

Example 2.3. The following process interacts on session $s$ using channels with role $s[s]$, $s[c]$, $s[a]$, to play resp. roles $s$, $c$, $a$. For brevity, we omit irrelevant message payloads.

$P_s = s[s][c]@cancel$

$(vs)(P_s \mid P_c \mid P_a)$

where:

- $P_c = s[c][s][\sum\{login.s[c][a]@passwd("XYZ")\}, cancel.s[c][a]@quit]$
- $P_a = s[a][c][\sum\{passwd(y).s[a][s]\}@$auth(y = "secret"), quit}$

Here, $(vs)(P_s \mid P_c \mid P_a)$ is the parallel composition of processes $P_s$, $P_c$, $P_a$ in the scope of session $s$. In $P_s$, “$s[s][c]@cancel” means: use $s[s]$ to send cancel to $c$. Process $P_c$ uses $s[c]$ to receive login or cancel from $s$; then, in the first case it uses $s[c]$ to send passwd to $a$; in the second case, it uses $s[c]$ to send quit to $a$. By Def. 2.2, we have the reductions:

$(vs)(P_s \mid P_c \mid P_a) \rightarrow (vs)(0 \mid s[c][a]@quit \mid P_a) \rightarrow (vs)(0\mid0\mid0) \equiv 0$

2.2 Types, Subtypes, and Typing

Session types (Def. 2.4) describe the intended use of communication channels in the MPST $\pi$-calculus (Def. 2.1); channels are mapped to their respective type by session typing contexts (Def. 2.6).

Definition 2.4. The syntax of multiparty session types is:

$S, T ::= p &_{i\in I} m_i(S_i) . S'_i \mid p \oplus_{i\in I} m_i(S_i) . S'_i \mid \text{end} \mid \mu t . S \mid t$ with $I \neq \emptyset$, and $m_i$ pairwise distinct

We require types to be closed, and recursion variables to be guarded.

The branching type (or external choice) $p &_{i\in I} m_i(S_i) . S'_i$ says that a channel must be used to receive from $p$ one input of the form $m_i(S_i)$, for any $i \in I$ chosen by $p$, where $m_i$ are message labels and $S_i$ are message payload types; then, the channel must be used following the continuation type $S'_i$. The selection type (or internal choice) $p \oplus_{i\in I} m_i(S_i) . S'_i$, instead, requires to use a channel to perform one output $m_i(S_i)$ towards $p$, for some $i \in I$, and continue using the channel according to $S'_i$. Type end describes a terminated channel allowing no further inputs/outputs. Type $\mu t . S$ models recursion: $\mu$ binds the recursion variable $t$ in $S$. The guardedness requirement ensures
We write composition using the "channel-oriented" order of Gay and Hole [2005]; Scalas et al. [2017a]. For a thorough comparison, see [Gay 2016].

Fig. 2. Multiparty session typing rules. Rule \(\text{T-v}\) for session restriction is discussed in §2.3.

that recursive types are contractive: i.e., in \(\mu S\) we have \(S \neq t'\) for all \(t'\). For brevity, we often omit the trailing end in types, and end-typed message payloads: e.g., \(p \odot m\) stands for \(p \odot m(\text{end})\). end.

In Def. 2.5 below, we define the multiparty session subtyping relation [Dezani-Ciancaglini et al. 2015].\(^1\) Intuitively, Def. 2.5 says that a type \(S\) is smaller than \(S'\) when \(S\) is "less demanding" than \(S'\)—i.e., when \(S\) imposes to support less external choices and allows to perform more internal choices. Session subtyping is used in the type system to augment its flexibility.

**Definition 2.5.** The session subtyping \(\leq\) is coinductively defined:

\[
\begin{align*}
\forall i \in I & \quad S_i \leq T_i & S'_i \leq T'_i & \text{[Sub-\&]} & \quad \forall i \in I & \quad T_i \leq S_i & S'_i \leq T'_i & \text{[Sub-\&]} \\
p \& \forall i \in m_i(S_i). S'_i \leq p \& \forall i \in m(T_i). T'_i & \text{[Sub-end]} & p = T \leq T \{ T \} & \text{[Sub-\&L]} & S \leq T \{ T \} & \text{[Sub-\&R]} \\
\end{align*}
\]

In Def. 2.5, rules [Sub-\&]/[Sub-\&] define subtyping on branch/select types: [Sub-\&] is covariant in both the carried types and in the number of branches, whereas [Sub-\&] is contravariant in both: this formalises the intuition of a smaller type having less external choices, and more internal choices. By rule [Sub-end], end is only subtype of itself. The recursion rules [Sub-\&L]/[Sub-\&R] relate types up-to their unfoldings, as usual for coinductive subtyping [Pierce 2002, Ch. 21].

**Definition 2.6 (Typing Contexts).** \(\Theta\) denotes a partial mapping from process variables to \(n\)-tuples of types, and \(\Gamma\) denotes a partial mapping from channels to types, defined as:

\[
\Theta := 0 \mid \Theta, X:S_1, \ldots, S_n \quad \Gamma := 0 \mid \Gamma, x:S \mid \Gamma, s[p]:S
\]

The composition \(\Gamma_1, \Gamma_2\) is defined iff \(\text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) = \emptyset\).

We write \(s \leq \Gamma\) iff \(\forall \rho : s[p] \in \text{dom}(\Gamma)\) (i.e., session \(s\) does not occur in \(\Gamma\)).

We write \(\text{dom}(\Gamma) = \{s\}\) iff \(\forall c \in \text{dom}(\Gamma)\) there is \(p\) such that \(c = s[p]\) (i.e., \(\Gamma\) only contains session \(s\)).

We write \(\Gamma \subseteq \Gamma'\) iff \(\text{dom}(\Gamma) = \text{dom}(\Gamma')\) and \(\forall c \in \text{dom}(\Gamma)\) : \(\Gamma(c) \subseteq \Gamma'(c)\).

\(^1\)Our \(\leq\) is inverted w.r.t. the “process-oriented” subtyping of Dezani-Ciancaglini et al. [2015] because, for convenience, we use the “channel-oriented” order of Gay and Hole [2005]; Scalas et al. [2017a]. For a thorough comparison, see [Gay 2016].
The type system uses two kinds of typing contexts: \( \Theta \) to assign an \( n \)-tuple of types to each process variable \( X \) (one type per argument), and \( \Gamma \) to map variables and channels with roles to session types. Together, they are used in judgements of the following form:

\[
\Theta : \Gamma \vdash P \quad \text{(with } \Theta \text{ omitted when empty)}
\]

meaning: “given the process types in \( \Theta \), \( P \) uses its variables and channels linearly according to \( \Gamma \).”

The typing judgement (5) is inductively defined by the rules in Fig. 2. For convenience, we type-annotate channels bound by process definitions and restrictions.

The first three rules in Fig. 2 define auxiliary judgements. By \([T-X]_1\), \( \Theta \vdash X : S_1, \ldots, S_n \) holds if \( \Theta \) maps \( X \) to an \( n \)-tuple of types \( S_1, \ldots, S_n \). By \([T-Sub]_1\), \( \Gamma \vdash c : S' \) holds if \( \Gamma \) only contains one entry \( c : S \) with \( S \leq S' \); i.e., when typing processes, \([T-Sub]\) allows to use a channel of type \( S \) whenever a channel with a larger type \( S' \) is needed, as per Liskov and Wing [1994]’s substitution principle; note that Def. 2.5 relates types up-to unfolding, hence \([T-Sub]\) makes the type system equi-recursive [Pierce 2002, Ch. 21]. Finally, \( \text{end}(\Gamma) \) holds if \( \Gamma \)’s entries are end-typed (under \([T-Sub]\)).

The other rules in Fig. 2 define the process typing judgement in (5). The termination rule \([T-\emptyset]\) says that \( \emptyset \) is typed if all channels in \( \Gamma \) are end-typed. By the process definition rule \([T-def]\), \( \text{def } X(x) = P \) in \( \Phi \) is typed if \( P \) uses the arguments \( x_1, \ldots, x_n \) according to \( S_1, \ldots, S_n \), and the latter is the type of \( X \) when typing both \( P \) and \( \Phi \); this means that \( P \) can refer to \( X \), and this allows to type recursive processes. By the process call rule \([T-X]\), \( X(\overline{x}) \) is typed if the types of \( \overline{x} \) match those of the formal parameters of \( X \), and any unused channel (in \( \Gamma_0 \)) is end-typed: this preserves linearity by ensuring that channels requiring more inputs/outputs cannot be forgotten. By the branching rule \([T-\oplus]\), \( \sum_{i \in I} \alpha_i m_i(y_i) \cdot P_i \) is typed if \( c \) has type \( S \), where \( S \) is an external choice from \( q \), with the same branching labels \( m_i \). The selection rule \([T-\sqcap]\) says that \( \alpha \sqcap \oplus m \cdot P \) is typed if \( c \) has type \( S \), where \( S \) is an internal choice towards \( q \) with message label \( m \). By the parallel rule \([T-\parallel]\), two parallel processes are typed by splitting the context in the premises. The session restriction rule \([T-s]_\nu\) deserves special attention: we discuss it in §2.3.

Example 2.7. Take the processes from Ex. 2.3, and the types \( S_s, S_c, S_a \) from §1, eq. (2). With the rules in Fig. 2, we have the following typing derivation:

\[
\begin{array}{c}
\vdash s[s]:S_s + P_s \\
\vdash s[c]:S_c + P_c
\end{array}
\]

\[
\begin{array}{c}
\vdash s[s]:S_s, s[c]:S_c + P_s \mid P_c \\
\vdash s[a]:S_a + P_a
\end{array}
\]

\[
\begin{array}{c}
\vdash \Gamma \vdash P_s \mid P_c \mid P_a
\end{array}
\]

where \( \Gamma = s[s]:S_s, s[c]:S_c, s[a]:S_a \)

The process \( P_s \mid P_c \mid P_a \) is typed by rule \([T-\parallel]\), that splits the typing context linearly ensuring that a channel is not used by two parallel sub-processes. In the omitted part of the derivation, processes \( P_s, P_c, P_a \) are typed separately, using rules \([T-\emptyset]/[T-\oplus]\): each process uses one of the channels with role \( s[s], s[c], s[a] \), according to the type \( S_s, S_c, S_a \), respectively.

We conclude with the transitions/reductions of typing contexts (Def. 2.8): intuitively, they abstract the message exchanges that might occur over typed channels. We adopt a standard formulation, with two adaptations: we compare payloads using \( \preceq \) (to cater for subtyping), and we specify transition labels for inputs, outputs, and communication.

\[ \frac{\alpha}{\text{typing context transition}} \]

\[ \text{Definition 2.8. Let } \alpha \text{ have the form } s[p \& q \cdot m](S), \text{ or } s[p \& q \cdot m](S), \text{ or } s[p, q \cdot m] \text{ (for any roles } p, q, \text{ message label } m, \text{ and type } S). \text{ The typing context transition } \rightarrow^{\alpha} \text{ is inductively defined by the rules:} \]
We write \( s \) to achieve type safety, we want the process in (7) to be untypable — which means that, when

\[
\begin{array}{c}
\Gamma \vdash s[p] : q & \vartriangleleft m(S_i) & S_i' \rightarrow S_k' \\
\Gamma_1 \xrightarrow{s[p] \vartriangleleft q} \Gamma_1' & \Gamma_2 \xrightarrow{s[q] \vartriangleleft m(T)} \Gamma_2' & S \leq T \\
\Gamma_1, \Gamma_2 \xrightarrow{s[p] \vartriangleleft q} \Gamma_1', \Gamma_2' \\
\end{array}
\]

We write \( \Gamma \xrightarrow{s[p] \vartriangleleft q} \Gamma' \) for some \( \Gamma' \). The reduction \( \Gamma \rightarrow \Gamma' \) is defined iff \( \Gamma \rightarrow^* \Gamma' \) for some \( s, p, q, m \). We write \( \Gamma \rightarrow^* \Gamma' \) for some \( \Gamma' \), and \( \Gamma \not\rightarrow^* \) for its negation (i.e., when there is no \( \Gamma' \) such that \( \Gamma \rightarrow \Gamma' \)). We define \( \rightarrow^* \) as the reflexive and transitive closure of \( \rightarrow \).

By \([\Gamma\vartriangleright] / [\Gamma\&] \) in Def. 2.8, a typing context entry can transition to one of its continuations by firing an output label of the form \( s[p] \vartriangleleft q \vartriangleright m(S) \) (in case of selection types), or an input label of the form \( s[p] \vartriangleleft q \vartriangleright m(S) \) (in case of branching types). Rule \([\Gamma\&\text{-Comm}]\) models type-level communication: e.g., it allows two entries \( s[p] : S_p, s[q] : S_q \) to interact, provided that: (1) \( S_p \) is a selection towards \( q \) (with a corresponding output transition); (2) \( S_q \) is a branching from \( p \) (with a corresponding input transition); and (3) they are firing a common message label \( m \), and the carried type \( S \) sent by \( S_q \) is subtype of the type \( T \) expected by \( S_p \). When all such conditions hold, \( s[p] : S_p, s[q] : S_q \) transition to the respective continuations, by firing a communication label \( s[p] \vartriangleleft q \vartriangleright m \) that records the session \( s \), and the message sender \( p \), recipient \( q \), and label \( m \) (the payload types are discarded).

In the rest of the paper, we will mostly use the unlabelled reduction \( \Gamma \rightarrow \Gamma' \), which means that \( \Gamma \) transitions to \( \Gamma' \) through some communication. The labelled transitions will be reprised in §5.

### 2.3 Towards Subject Reduction and Type Safety

In §1, we mentioned that a process naively typed with an arbitrary \( \Gamma \) can “go wrong.” Indeed, by themselves, the typing rules in Fig. 2 do not guarantee type safety, as shown by the following (counter-)example:

\[
\begin{array}{l}
\vdash s[p] : q \vartriangleleft \text{foo}(\text{end}), \ s[q] : p & \vartriangleright \text{bar}(\text{end}), \ s'[r] : \text{end} \quad \vdash s[p] : q \vartriangleleft \text{foo}(s'[r]) \quad \vdash s[q] : p \sum \text{bar}(x) \rightarrow \text{err} \quad (6)
\end{array}
\]

Intuitively, the problem of this typing judgement can be seen in its typing context: the type of \( s[p] \) outputs \( \text{foo} \) to \( q \), but the type of \( s[q] \) expects \( \text{bar} \). This means that we need a criterion to reject (6).

Importantly, the same criterion must be applied for **typing session restriction**. Consider rule [\( T-v \)] in Fig. 2: it types a restricted session \( s \) with \( \Gamma' \), provided that (1) \( \Gamma' \) only contains channels with roles belonging to \( s \); (2) the restricted \( s \) does not occur in the remaining context \( \Gamma \) (to avoid clashes); and (3) \( \Gamma' \) satisfies a (yet unspecified) property \( \varphi \). How should we define \( \varphi \)? It cannot be always true, because we would have this counterexample to type-safety, where \( \Gamma \) is the context in (6):

\[
\emptyset \vdash (vs : \Gamma) \left( s[p] : q \vartriangleleft \text{foo}(s'[r]) \right) \quad s[q] : p \sum \text{bar}(x) \rightarrow \text{err} \quad (6) \quad \text{by (6) and rule [\text{R-Crx}] in Fig.1} \quad (7)
\]

To achieve type safety, we want the process in (7) to be untypable — which means that, when type-checking \( (vs : \Gamma) \ldots \), we must ensure that \( \varphi \) in rule [\( T-v \)] does not hold for \( \Gamma \), in cases like (6).

Moreover, \( \varphi \) must be technically usable to prove subject reduction; this leads to three **desiderata**:

**D1** \( \varphi \) must make the typing context “safe”: if the type of \( s[p] \) sends a message to \( q \), then the type of \( s[q] \) must be able to input such a message;

**D2** \( \varphi \) must be preserved when the typing rule [\( T-\ast \)] splits typing contexts (see derivation in Ex.2.7);

**D3** \( \varphi \) must be preserved when processes, and typing contexts, interact and reduce (Def. 2.2/2.8).

Therefore, the choice of the criterion for handling cases like (6) has a deep impact on the theoretical foundations of the type system: it determines how subject reduction and type safety properties are stated and proved, and how general/restrictive they are; it also determines how to define \( \varphi \) in rule [\( T-v \)], to correctly type session restriction \( (vs) P \), and handle cases like (7).
In §4, we show how our new MPST theory establishes its foundations, and \( \phi \) in rule \([T-v] \). But first, in §3, we show how such choices are made in classic MPST, and what are the consequences.

3 LIMITATIONS AND THEORETICAL ISSUES OF CLASSIC MPST

This section gives a formal basis to our claims in §1: in §3.1 we use our opening example to show the technical issues of classic MPST, caused by consistency (also called coherency, e.g., by Deniélou et al. [2012]); and in §3.2, we provide further examples that are rejected by classic MPST. Our new MPST system (§4) eschews these problems, by adopting a more general theoretical basis.

Remark 3.1. The issues described in this section do not apply to two recent MPST works, by Dezani-Ciancaglini et al. [2015] and Scalas and Yoshida [2018]: they have different, non-classic MPST theories. However, such works have other limitations, surmounted by this paper: they are detailed in §8.2.

3.1 Consistency and Subject Reduction

To reject cases like (6) (§2.3), classic MPST require typing contexts to be consistent: for each pair of entries \( \{s[p]:S_p, s[q]:S_q\} \subseteq \Gamma \), the inputs/outputs of \( S_p \) from/to \( q \) must be dual w.r.t. the outputs/inputs of \( S_q \) to/from \( p \). This guarantees that two roles \( p, q \) can only send/receive compatible messages in a session \( s \). More precisely, consistency requires to check the duality of the partial projections \( S_p|q \) and \( S_q|p \), using Def. 3.5, 3.6, 3.7, and 3.8 (collected in Fig.3): this clearly shows that MPST were developed by adopting a proof framework based on binary session types.

Correspondingly, to reject cases like (7), classic MPST define rule \([T-v] \) in Fig.2 by setting \( \phi = \) consistent. This yields the classic session restriction typing rule:

\[
\Gamma' = \{s[p]:S_p\}_{p \in I} \quad s \notin \Gamma \quad \text{consistent}(\Gamma') \quad \Theta \cdot \Gamma, \Gamma' \vdash P \\
\Theta \cdot \Gamma \vdash (\nu s:\Gamma') P
\]

and this is sound (indeed, consistency satisfies the desiderata (D1)–(D3) described in §2.3).

E.g., the typing context in (6) is not consistent; correspondingly, no consistent \( \Gamma \) can be assigned to (vs) ... in (7): hence, with rule \([T-v] \), the process in (7) is untypable in classic MPST.

Limitations of Consistency. Take the processes from Ex.2.3, and the typing derivation from Ex.2.7. Using the rules in Fig.2 with \([T-v] \) above, we might try to type our opening example as:

\[
\begin{align*}
\Gamma \quad \text{consistent} \\
\vdash \Theta \cdot (\nu s:\Gamma) (P_s | P_c | P_a) \\
\Theta \vdash (\nu s:\Gamma) (P_s | P_c | P_a)
\end{align*}
\]

As shown in §1(2), the types \( S_s, S_c, S_a \) assigned to \( s[s], s[c], s[a] \) are respectively \( G|s, G|c, G|a \), i.e., the projections of \( G \) (Def. 3.3). However, the derivation in (8) is wrong, because the consistency premise of \([T-v] \) does not hold. To see why, we need to check all pairs of types for session \( s \):

- \( S_s \) are consistent: the outputs of \( S_s \) to \( c \) are dual w.r.t. the inputs of \( S_c \) from \( s \);
- \( S_a \) are not consistent, because the partial projections \( S_a|a \) and \( S_a|s \) are undefined (Def. 3.6).

Intuitively, \( S_a|a \) and \( S_a|s \) are undefined because the inputs/outputs of \( S_s|S_a \) from/to \( a/s \) depend on previous I/O with \( c \): i.e., if the service \( s \) sends login (resp. cancel) to the client \( c \), then \( s \) will (resp. will not) later interact with the authorisation server \( a \). This is not captured by the syntactic nature of projection/duality checks: i.e., protocols with inter-role dependencies are often not consistent — even simple ones, like \( G \) in (1). Consequently, the process in Ex.2.3 is untypable, albeit correct (does not reduce to \( \text{err} \)).
Definition 3.2. The syntax of a global type $G$ is:
\[
G ::= \text{p} \rightarrow q \cdot \{m_i(S_i) : G_i \}_{i \in I} \mid \mu t.G \mid t \end{equation}
\]
with $\text{p} \neq q$, $I \neq \emptyset$, and $\forall i \in I : \text{fv}(S_i) = \emptyset$

We write $\text{p} \in \text{roles}(G)$ (or simply $\text{p} \in G$) iff, for some $q$, either $\text{p} \rightarrow q$ or $q \rightarrow \text{p}$ occurs in $G$.

Definition 3.3 (Global Type Projection). The projection of $G$ onto $\text{p}$, written $G[\text{p}]$, is:
\[
(q \rightarrow r : \{m_i(S_i) : G_i \}_{i \in I})[\text{p}] = \begin{cases}
\text{r} \pi_{i \in I} m_i(S_i).G_i[\text{p}] & \text{if } \text{p} = \text{q} \\
\text{q} \pi_{i \in I} m_i(S_i).G_i[\text{p}] & \text{if } \text{p} = \text{r} \\
\bigcap_{i \in I} G_i[\text{p}] & \text{if } \text{q} \neq \text{p} \neq \text{r}
\end{cases}
\]

Definition 3.4 (Partial Session Types). Partial session types, ranged over by $\mu$, are:
\[
\forall i \in I \exists m_i(S_i).H_i = \bigcap_{i \in I} m_i(S_i).H_i \mid t \ \text{end}
\]

Definition 3.5 (Duality of Partial Session Types). The dual of $H$, written $\overline{H}$, is:
\[
\forall i \in I m_i(S_i).H_i = \bigcap_{i \in I} m_i(S_i).H_i \mid t \ \text{end}
\]

Definition 3.6 (Partial Projection). The projection of $S$ onto $\text{p}$, written $S[\text{p}]$, is:
\[
(q \& \pi_{i \in I} m_i(S_i).S_i')[\text{p}] = \begin{cases}
\bigcap_{i \in I} S_i'[\text{p}] & \text{if } \text{p} = \text{q} \\
\pi_{i \in I} m_i(S_i).S_i'[\text{p}] & \text{if } \text{p} \neq \text{q}
\end{cases}
\]

Definition 3.7. Subtyping for partial types is coinductively defined (we omit unfolding rules, cf. Def. 2.5):
\[
\forall i \in I : S_i \leq T_i \leq H_i' \iff \forall i \in I \exists m_i(S_i).H_i' = \bigcap_{i \in I} m_i(S_i).H_i' \mid t \ \text{end}
\]

Definition 3.8. $\Gamma$ is consistent iff, $\forall S, \text{p} \neq q, S, T, \{s[p] ; S, s[q] ; T\} \subseteq \Gamma$ implies $\overline{S[q]} \leq T[\text{p}]$.

Fig. 3. Classic MPST: global types, projections, consistency, and duality. Note that all these definitions are not necessary in our new theory of multiparty session types (§4).

Subject Reduction and Type Safety (or Lack Thereof). As noted in §1, the classic MPST subject reduction statement is (4). Now, consider (8) again: the conclusion is wrong, but the intermediate judgement $\Gamma \vdash P_s \mid P_c \mid P_a$ holds. For this judgement, the subject reduction statement (4) is vacuously true (since $\Gamma$ is not consistent): hence, we cannot prove that $P_s \mid P_c \mid P_a$ “never goes wrong.”

Interplay Between Consistency and Global Type Projection. The consistency requirement constrains the MPST theory in non-obvious ways, causing subtle issues with global type projections. Several
MPST papers claim that if \( \Gamma \) is obtained by projecting a global type \( G \), then \( \Gamma \) is consistent (see e.g.: [Deniélou et al. 2012, p.28], [Coppo et al. 2015a, Prop. 1], [Chen 2015, Prop. 2]). This claim corresponds to introducing the typing rule \( [T \vdash \text{vClassic \ G}] \) below, that seemingly fixes derivation (8):

\[
\begin{align*}
\Gamma' = \{s[p] : G[p] \}_{p \in \text{Roles}(G)} & \quad \Theta \cdot \Gamma \vdash P \\
\text{if } s \not\in \Gamma & \quad \Theta \cdot \Gamma, \Gamma' \vdash P
\end{align*}
\]

Unfortunately, our example in §1 shows a global type whose projections are not consistent. This is because we use the “full merging” projection (Def. 3.3), introduced in Deniélou et al. [2012]; Yoshida et al. [2010] to type more processes. The intuition is the following. Take the initial choice of the global type \( G \) in §1(1) (reported below), that does not involve role \( a \):

\[
G = s \rightarrow c : \{\text{login}.G_1, \text{cancel}.G_2\}
\]

where \( G_1 = c \rightarrow a : \text{passwd}(\text{Str}) \cdot a \rightarrow s : \text{auth}(\text{Bool}) \)

\( G_2 = c \rightarrow a : \text{quit} \)

To project \( G \) onto \( a \), we must “skip” the first interaction between \( s \) and \( c \), and merge the projections of \( G_1 \) and \( G_2 \) onto \( a \), rejecting potentially unsafe local types combinations (thus avoiding cases like (6) above). Consequently, projection works as follows:

\[
G | a = S_1 \cap S_2
\]

where \( S_1 = G_1 | a = c \rightarrow a : \text{passwd}(\text{Str}) . s @ \text{auth}(\text{Bool}) \)

\( S_2 = G_2 | a = c \rightarrow a : \text{quit} \)

We now have two possibilities, depending on how we choose the merging operator \( \sqcap \) (Def. 3.3):

- **plain merging**: \( S_1 \cap S_2 = S_1 \) iff \( S_1 = S_2 \) (undefined otherwise);
- **full merging**: \( S_1 \cap S_2 = S_a \) (see (2) in §1).

i.e., the restrictive plain merging is undefined for our example \( G \), while full merging yields all desired projections — but they are not consistent, as shown above. Consequently, the tentative rule \( [T \vdash \text{vClassic \ G}] \) with “full merging” projections breaks subject reduction proofs. E.g., take \( P \) typed by \( [T \vdash \text{vClassic \ G}] \), and reducing to \( P' \), as follows:

\[
\Theta \cdot \Gamma \vdash P \quad \text{with } P = (v : \Gamma) P_0 \rightarrow (v : \Gamma') P_1 = P' \quad \text{(induced by } P_0 \rightarrow P_1 \text{ and rule } \text{[R-Ctx] in Fig.1)} \]

(9)

To prove subject reduction as stated in (4), we need to invert \( P \)’s typing and apply the induction hypothesis on \( \Theta \cdot \Gamma \vdash P_0 \) and \( P_0 \rightarrow P_1 \) (from (9)), to obtain that there is some \( \Gamma' \) such that \( \Gamma \rightarrow \Gamma' \) and \( \Theta \cdot \Gamma' \vdash P_1 \); however, to apply (4) in the induction hypothesis we need \( \Gamma \) consistent, and we have shown that this hypothesis might not hold.

We can now revisit our claims in §1, making them precise, and highlighting the resulting impasse:

**(C1) overly restrictive**: requiring \( \Gamma \) consistent drastically constrains typability: it rejects our simple example in §1, and many other correct protocols (see §3.2 later on). Correspondingly, the restrictive “plain merging” projection of [Honda et al. 2008, Def. 4.1] and [Coppo et al. 2015a, Def. 1], guarantees consistency by rejecting many correct protocols;

**(C2) inflexible and error-prone**: if we use a “full merging” projection as in, e.g., Yoshida et al. [2010] or Deniélou et al. [2012], then \( \Gamma \) might not be consistent. This means that the proofs of subject reduction depending on “full merging” (e.g. [Yoshida et al. 2010, Thm 3.5], [Deniélou et al. 2012, Thm 4.6]), and successive papers discussed in §8) do not work; we might fix such proofs by adding a consistency requirement — but then, we would fall back into (C1) above.

In §4, we completely eschew these issues by developing new theoretical foundations for MPST: we cut the ties with binary session types, adopting a more general, behavioural safety invariant, that subsumes consistency and binary session duality.

### 3.2 More Examples of Correct, yet Non-Consistent Protocols

We conclude this section with Fig. 4, that describes various multiparty protocols, formalised as typing contexts. None of such protocols is consistent, because some of their partial projections are
We now present our new general MPST theory. Its generality comes from the fact that it is based on general type system (§4); moreover, they have different behavioural properties, analysed in §5.3 (Table 1).

Fig. 4. A selection of multiparty protocols: each one is expressed as a (non-consistent) typing context (on the right); for the sake of clarity, we also outline the shape of a global type with corresponding projections (on the left). The exception is protocol (2), that cannot be projected from any global type: see §3.2. Being non-consistent, all these protocols are not supported by classic MPST — but they are all supported by our new general type system (§4); moreover, they have different behavioural properties, analysed in §5.3 (Table 1).

undefined — as a consequence of the issues illustrated in §3.1; moreover, the protocols (2), (3) and (4) trigger further subtle restrictions in the partial projection/merging of recursive types (Def. 3.6).

Notably, Fig. 4 includes an example of multiparty protocol that cannot be projected from any global type: the recursive two-buyers protocol (2). The key issue is in the type of s[a], when alice interacts with bob: alice sends a message to the store in one of the branches under recursion μt....(where bob answers yes), but not in the other branch (where bob says no). This is not supported by projection and merging (Def. 3.3): they can only generate session types where all branches under recursion syntactically contain a same set of roles. Consequently, no global type can be projected and yield the type of s[a] in Fig. 4(2). This restriction does not impact our new MPST theory (§4).

4 A NEW, GENERAL MULTIPARTY SESSION TYPE SYSTEM

We now present our new general MPST theory. Its generality comes from the fact that it is based on a weak typing context safety invariant, that rejects cases like (6)/(7) (§2.3) without the restrictions
and drawbacks of classic MPST consistency. Moreover, we design the new type system to be parametric on the safety invariant itself: by fine-tuning the parameter, the type system can accept or reject MPST processes depending on the properties of the protocols they implement (we will take advantage of this feature in §3). Hence, different instantiations of the parameter yield different type system instances — but we just need to prove type safety once, under the weakest safety invariant. This design is inspired by Igarashi and Kobayashi [2004]’s Generic Type System for the \( \pi \)-calculus.

We first formalise what a “safety invariant” is, in Def. 4.1 below: it is a behavioural property of typing contexts, that depends on how they reduce (cf. Def. 2.8). The fundamental difference with classic MPST (§3) is that our safety is not based on binary session types, nor duality.

**Definition 4.1.** \( \phi \) is a safety property of typing contexts iff:

- \([\text{}-\&] \phi(\Gamma, s[p]:q\oplus_{i\in I} m_i(S_i), \cdot, S', s[q]:p\&_{j\in J} m_j(T_j), \cdot \rightarrow) \) implies \( I \subseteq J \), and \( \forall i \in I : S_i \subseteq T_i \);
- \([\text{}-\mu] \phi(\Gamma, s[p]:\mu t. S) \) implies \( \phi(\Gamma, s[p]:\mu t. S \{ t. S / t \}) \);
- \([\text{}-\to] \phi(\Gamma) \) and \( \Gamma \to \Gamma' \) implies \( \phi(\Gamma') \).

We say \( \Gamma \) is safe, written safe(\( \Gamma \)), if \( \phi(\Gamma) \) for some safety property \( \phi \).

The rules of Def. 4.1 directly satisfy the desiderata (D1) and (D3) discussed in §2.3 (whereas (D2) is satisfied by Lemma 4.3, as we will see shortly). Rule \([\text{}-\&\&] \) says that the roles in a safe typing context can only exchange compatible messages (this is desideratum (D1)); more precisely, if the typing context contains entries for \( s[p] \) and \( s[q] \), with \( p \) sending to \( q \) and \( q \) receiving from \( p \), then \( p \) support all \( q \)'s messages — and thus, they can reduce, by Def. 2.8. Rule \([\text{}-\mu] \) says that \( \phi \) contains all recursive type unfoldings: this allows rule \([\text{}-\&\&] \) to check unfolded types, where \( \oplus \& \) occur at the top-level. By rule \([\text{}-\to] \), safety is preserved whenever \( \Gamma \) reduces (this is desideratum (D3)).

**Example 4.2.** The typing context \( \Gamma \) of (8) in §3 is safe. This can be easily verified by: (1) defining \( \phi \) as \( \phi = \{ \Gamma' | \Gamma \to^* \Gamma' \} \), i.e., containing \( \Gamma \) and all its reductions; (2) checking that \( \phi \) is a safety property, because all its elements satisfy the clauses of Def. 4.1; and (3) concluding that, since \( \phi(\Gamma) \) holds, \( \Gamma \) is safe. Instead, the typing context in (6) is not safe: any property \( \phi \) containing such typing context is not a safety property, as it violates clause \([\text{}-\&\&] \) of Def. 4.1.

Def. 4.1 also has the properties below, useful for proving subject reduction: typing context splits preserve safety (Lemma 4.3, which satisfies the remaining desideratum (D2) in §2.3); if \( \Gamma \) is safe, then supertyping/reductions commute (Lemma 4.4); supertyping preserves safety (Lemma 4.5).

**Lemma 4.3.** If \( \Gamma, \Gamma' \) is safe, then \( \Gamma \) is safe.

**Lemma 4.4.** If \( \Gamma \) safe and \( \Gamma \preceq \Gamma' \to \Gamma'' \), then there is \( \Gamma''' \) such that \( \Gamma \to \Gamma''' \preceq \Gamma'' \).

**Lemma 4.5.** If \( \Gamma \) is safe and \( \Gamma \preceq \Gamma' \), then \( \Gamma' \) is safe.

We can now define our new multiparty session type system. As explained in §2.3, since we are adopting safety (Def. 4.1) as the criterion for accepting/rejecting typing contexts, we use the same criterion to define a typing rule for session restriction.

**Definition 4.6 (General Multiparty Session Type System).** The general MPST typing judgement is inductively defined by the rules in Fig. 2 — with rule \([\text{}-\cap] \) restricted as follows:

\[
\frac{\Theta \cdot \Gamma \vdash P}{\Theta \cdot \Gamma \vdash (vs: \Gamma) P} \quad [\text{\text{GEN-\cap}}]
\]

where \( \phi \) is a safety property.

Given a safety property \( \phi \), we write “\( \Theta \cdot \Gamma \vdash P \) with \( \phi \)” to instantiate \( \phi \) in \([\text{\text{GEN-\cap}}] \) above; when “with \( \phi \)” is omitted, the instantiation is \( \phi = \text{safe} \) (i.e., the largest safety property, cf. Def. 4.1).
Example 4.7. Take the (wrong) typing derivation (8) in §3.1, and replace the (wrong) application of rule $[\text{?}\vdash \text{?CLASSIC}]$ with $[\text{?Gen-?}]$ from Def. 4.6, instantiating $\varphi$ with the safety property of Ex. 4.2 (that contains $\Gamma$). The resulting typing derivation is correct.

Ex. 4.7 above shows that our new type system is not limited by consistency requirements, and types our opening example. Notably, the only visible difference between our new type system (Def. 4.6) and the classic one (§3.1) is that $[\text{?Gen-?}]$ uses a (parametric) safety property $\varphi$, instead of consistency. As explained in §2.3, this small visible difference between typing rules is a manifestation of a deeper underlying change: by removing the crucial consistency/duality assumption of classic MPST, we are replacing its theoretical underpinnings, and this requires a revision of all MPST soundness proofs. The payoff is that our new MPST theory enjoys a much more general subject reduction property (Thm. 4.8, based on Lemmas 4.3 to 4.5); from this, we get that typed processes “never go wrong” (Cor. 4.9). And again, unlike classic MPST, these results are not limited by consistency.

Theorem 4.8 (Subject Reduction). Assume $\Theta \cdot \Gamma \vdash P$ and $\Gamma$ safe. Then, $P \rightarrow P'$ implies $\exists \Gamma'$ safe such that $\Gamma \rightarrow^* \Gamma'$ and $\Theta \cdot \Gamma' \vdash P'$.

Corollary 4.9 (Type safety). If $\Theta \cdot \Gamma \vdash P$ and $P \rightarrow^* P'$, then $P'$ has no error.

Proof. We first prove a more general result. Assume $\Theta \cdot \Gamma \vdash P$ with $\Gamma$ safe, and $P = P_1 \rightarrow \cdots \rightarrow P_n = P'$. By induction on $n$, using Thm. 4.8, we prove $\Theta \cdot \Gamma' \vdash P'$, for some safe $\Gamma'$ such that $\Gamma \rightarrow^* \Gamma'$. Now, by contradiction, assume that $P'$ has an error (Def. 2.2); then, $P'$ is untypable, since its err subterm is untypable: contradiction. Hence, $P'$ has no errors. We obtain Cor. 4.9 as a special case of the result above, with $\Theta = \emptyset$ and $\Gamma = \Gamma' = \emptyset$ (that is vacuously safe).

Example 4.10. Take our opening example in §1, and its typed process from Ex. 2.7 and 4.7. Using our new Thm. 4.8 instead of the classic MPST subject reduction (4) in §1, we infer that all process reductions are well-typed. And by Cor. 4.9, we are guaranteed that they do not contain errors.

Finally, note that type checking is decidable, whenever Def. 4.6 is instantiated with a decidable safety property: this mainly follows because typing rules are syntax-directed, and for any $P$, at most one can be applied. Also note that, since we proved Thm. 4.8 and Cor. 4.9 using the largest (i.e., the weakest) safety property, we do not need to repeat the proof depending on how $\varphi$ is instantiated in Def. 4.6: subject reduction and type safety hold for any safety property $\varphi$.

Theorem 4.11. If $\varphi$ is decidable, then “$\Theta \cdot \Gamma \vdash P$ with $\varphi$” is decidable.

5 VERIFYING RUN-TIME PROPERTIES OF PROCESSES, USING TYPES

In this section, we show that by suitably instantiating $\varphi$ in our type system (Def. 4.6), we can statically enforce desired run-time properties on processes — e.g., deadlock freedom and liveness.

In order to achieve this result, we study several typing context properties, and compare them with safety (Def. 4.1). The main reason for this study is that safety, albeit guaranteeing error-freedom (Thm. 4.8, Cor. 4.9), is otherwise rather weak. E.g., the following typing context is safe but deadlocked (it cannot reduce, because $p$ is waiting an input from $q$, who is waiting for $r$, who is waiting for $p$):

$$s[p]\cdot q\&m_1 \cdot r\&m_2, \ s[q]\cdot r\&m_3, \ p\&m_1, \ s[r]\cdot p\&m_2, \ q\&m_3$$

and the context above types deadlocked processes that cannot reduce, either. This is undesirable: “real-world” programs should be deadlock-free, or even live (i.e., each pending input/output should be fired, eventually). Therefore, stronger typing context properties are needed — and in our new MPST theory, we can use the parameter $\varphi$ of Def. 4.6 to enforce them, without consistency limitations.

In §5.4, we show that all typing derivations of classic MPST are valid under Def. 4.6: consistency implies safety, hence in $[\text{?Gen-?}]$ we can let $\varphi = \text{consistent}$; and in §5.5, we show how $\varphi$ statically determines the run-time properties on processes.
We first discuss several desirable, although undecidable, run-time properties of processes, such as deadlock-freedom and liveness (§5.1); next, we prove session fidelity, a crucial result that connects typing context reductions to processes reductions (§5.2). Then, we present various typing context properties (§5.3), and compare them (§5.4); finally, we show that they are decidable, and, with our new type system, they can be used to ensure that processes are, e.g., deadlock-free and live (§5.5).

5.1 Run-Time Properties of Processes

In Def. 5.1 below, we formalise various desirable process properties. All these properties are undecidable, because the MPST π-calculus is Turing-powerful [Busi et al. 2009]. To surmount this obstacle, from §5.3 we will reason on analogous properties for types (that are not Turing-powerful).

**Definition 5.1 (Process properties).** *P* is **deadlock-free** iff \( P \rightarrow^* P' \not\leftrightarrow \) implies \( P' \equiv 0 \). *P* is **terminating** iff it is deadlock-free, and \( \exists j \) finite such that, \( \forall n \geq j, P = P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_n \) implies \( P_n \equiv 0 \). *P* is **never-terminating** iff \( P \rightarrow^* P' \) implies \( P' \rightarrow^* \). *P* is **live** iff \( P \rightarrow^* P' \equiv \exists C[Q] \) implies:

1. if \( Q = c[q] \oplus m(s'[r]) \). \( Q' \) (for some \( m, s', r, Q' \)), then \( \exists C' : P' \rightarrow \exists C'[Q'] \); and
2. if \( Q = c[q] \sum_{i \in I}(x_i) \). \( Q' \) (for some \( m, x_i, Q' \)), then \( \exists C', k \in I, s', r : P' \rightarrow \exists C'[Q_k' \{ s'[r]/x_k \}] \).

*P* is **strongly live** iff \( P \rightarrow^* P' \equiv \exists C[Q] \) implies:

1. item 1 above, and moreover, there is \( n \) finite such that, whenever \( P' = P_0' \rightarrow P_1' \rightarrow \cdots \rightarrow P_n' \), then for some \( j \leq n \) we have \( P_j' \rightarrow \exists C'[Q'] \) (for some \( C' \));
2. item 2 above, and moreover, there is \( n \) finite such that, whenever \( P' = P_0' \rightarrow P_1' \rightarrow \cdots \rightarrow P_n' \), then for some \( j \leq n \) we have \( P_j' \rightarrow \exists C'[Q_k' \{ s'[r]/x_k \}] \) (for some \( C', k \in I, s', r \)).

In Def. 5.1, a process \( P \) is deadlock-free when it only stops reducing by becoming \( 0 \); \( P \) is terminating when it always reaches \( 0 \) after a finite number of reductions; \( P \) is never-terminating when it reduces forever; \( P \) is live (a.k.a. “lock-free” [Kobayashi and Sangiorgi 2010; Padovani 2014]) when all its pending inputs/outputs can always eventually communicate with a corresponding output/input; \( P \) is strongly live when all its pending inputs/outputs will always find a corresponding output/input, enabling communication after a finite number of reductions.

**Example 5.2.** We now illustrate the differences among the properties in Def. 5.1. Let:

\[
P = P_1 | P_2 \quad \text{where} \quad \begin{cases} \quad P_1 = s[p][q] \sum r \exp P \\
\quad P_2 = \text{def } X(x) = x[r] \sum \{ m_1.X(x) , m_2.x[p] \oplus r \exp 0 \} \quad \text{in } X(s[q]) \mid Q
\end{cases}
\]

i.e., \( P_1 \) implements \( p \), and waits a response from \( q \); \( P_2 \) implements \( q \), and loops every time role \( r \) (whose omitted implementation is in \( Q \)) sends \( m_1 \); if/when \( r \) chooses to send \( m_2 \), then \( P_2 \) sends the response to \( p \), triggering the input in \( P_1 \). Now, consider the following implementation of \( Q \):

\[
Q = \text{def } Y(y) = y[q] \oplus m_1.Y(y) \quad \text{in } Y(s[r])
\]

i.e., \( r \) sends \( m_1 \) to \( q \) forever — hence, \( P \) reduces forever, which means that \( P \) is never-terminating and deadlock-free. But note that the sub-process \( P_1 \) never has a chance to receive the desired response from \( q \); hence, \( P \) is not live. To address this, we can instead define \( Q \) above as:

\[
Q = s[r][q] \oplus m_1.s[r][q] \oplus m_2.0 | Q' \quad \text{where} \quad Q' = \begin{cases} \quad \text{def } Z(z) = z[r'[r']][m_3].Z(z) \quad \text{in } Z(s[r']) | Z'(s[r'[r']])
\end{cases}
\]

i.e., \( r \) sends \( m_1 \) and then \( m_2 \) to \( q \), and this causes \( q \) to send \( r' \) resp to \( p \) (cf. \( P_2 \) above); meanwhile, the sub-process \( Q' \) loops, with \( r' \) and \( r'' \) exchanging message \( m_3 \). With this definition of \( Q \), we obtain that \( P \) is live, because \( P_1 \) can always eventually receive its input while \( P_2 \) reduces.
Still, \( P \) is not strongly live, because the input of \( P_1 \) could be arbitrarily delayed by letting \( Q' \) reduce forever, without firing the outputs of \( Q \). We can make \( P \) strongly live, e.g., by redefining \( Q' \) as \( Q' = 0 \); this guarantees that \( P_1 \) will receive its input within 3 reductions.\(^3\)

### 5.2 Session Fidelity

We now prove that if a typing context can reduce, then a typed process \( P \) simulates the reduction (Thm. 5.4). A related result can be proved for classic MPST — but in our new theory, it is stronger: we do not assume consistency of the typing context, nor the existence of a global type projecting it. Session fidelity requires \( P \) to be (1) not deadlocked, and (2) \textit{productive}, i.e., not trapped in a loop like \( \text{def } X(x) = X(x) \text{ in } X(s[p]) \), if \( s[p] \) needs to be used for input/output: this is formalised in Def. 5.3.

**Definition 5.3.** Assume \( \emptyset \cdot \Gamma \vdash P \). We say that \( P \):

1. **has guarded definitions** iff in each subterm of the form \( \text{def } X(x_1:S_1, \ldots, x_n:S_n) = Q \text{ in } P' \), for all \( i \in 1 \ldots n \), \( S_i \not< \text{end} \) implies that a call \( \text{Y}(\ldots, x_i, \ldots) \) can only occur in \( Q \) as subterm of \( x_i[q]\sum_{j\in I}m_j(y_j) \cdot P \) or \( x_i[q] \oplus m(c).P'' \) (i.e., after using \( x_i \) for input/output);
2. **only plays role \( p \) in \( s \) by \( \Gamma \)** iff: (i) \( P \) has guarded definitions; (ii) \( \text{fv}(P) = \emptyset \); (iii) \( \Gamma = \Gamma_0, s[P]:S \) with \( S \not< \text{end} \) and end(\( \Gamma_0 \)); (iv) in all subterms (\( \nu' : \Gamma' \)) \( P' \) of \( P \), we have end(\( \Gamma' \)).

We say \( "P \) only plays role \( p \) in \( s" \) iff \( \exists \Gamma : \emptyset \cdot \Gamma \vdash P \), and item 2 holds.

We will explain item 1 of Def. 5.3 shortly (after Thm. 5.4). Item 2 identifies a process that plays exactly one role on one session: clearly, an ensemble of such processes cannot deadlock by waiting for each other on multiple sessions. All our examples (except a few, duly noted) satisfy Def. 5.3(2).

Now, in Thm. 5.4 we prove that a set of processes involved in a single session simulates the typing context, following its types/protocols. This addresses the typical application scenario of MPST: an ensemble of programs \( P_p \) interact on a multiparty session \( s \), each one playing a distinct role \( p \).

**Theorem 5.4 (Session Fidelity).** Assume \( \emptyset \cdot \Gamma \vdash P \), where \( \Gamma \) is safe, \( P \equiv \bigcup_{p \in P} P_p \), and each \( P_p \) either is \( \emptyset \) (up-to \( \equiv \)), or only plays \( p \) in \( s \). Then, \( \Gamma \vdash P' \) implies \( \exists \Gamma', P' \) such that \( \Gamma \vdash \Gamma' \), \( P \vdash P' \) and \( \emptyset \cdot \Gamma' \vdash P' \), where \( P' \equiv \bigcup_{p \in P} P'_p \) and each \( P'_p \) either is \( \emptyset \) (up-to \( \equiv \)), or only plays \( p \) in \( s \).

Note that in Thm. 5.4, \( P \) chooses which reduction of \( \Gamma \) to follow: in fact, a selection type in \( \Gamma \) might allow to choose \( m_1, \ldots, m_n \) (with different continuations), but \( P \) might select only one \( m_k \) (by \( \Gamma \cdot \oplus \)) in Fig. 2, and subtyping. This observation will be a crucial when reasoning about process liveness (§5.5). Also note that Thm. 5.4 relies on item 1 of Def. 5.3. In fact, by rule \( \Gamma \cdot \text{def} \) (Fig. 2), an unguarded definition \( X(x:S) = X(x) \) can be typed with any \( S \); therefore, we have e.g.:

\[
\emptyset \cdot s[p]:q \oplus m, s[q]:p \oplus m + \text{def } X(x:q \oplus m) = X(x) \text{ in } X(s[p]) \mid s[q][p] \sum_{m}
\]

and the unguarded process above reduces vacuously by calling \( X \) infinitely, without matching any typing context reduction; this explains the need of guarded definitions in Thm. 5.4.

### 5.3 Typing Context Properties

Fig. 5 lists several behavioural properties of typing contexts. In §5.5, we will show how they can statically enforce the run-time process properties discussed in §5.1.

- \( \Gamma \) is deadlock-free iff it stops reducing only when it only contains \( \text{ends} \);
- \( \Gamma \) is terminating iff it always reaches a final configuration, in a finite number of steps;
- \( \Gamma \) is never-terminating iff it never stops reducing;
- \( \Gamma \) is live, live’ or live” iff each branching/selection can be eventually fired.

\(^3\)As a more laborious alternative, we could formalise and assume a notion of fair scheduling, that eventually fires any action that is persistently enabled; we adopt a similar intuition for type reductions, in Def. 5.5.
Fig. 5. Properties of typing contexts. Each property is presented in two equivalent formalisations: the left-side ones are based on the notation and definitions introduced up to §5.4 (excluded); the right-side ones are μ-calculus formulas (explained in §6), and allow to verify typing contexts via model checking (e.g., with tools like mCRL2 [Groote and Mousavi 2014]).
The intuition behind live/live⁺/live++⁺ is the following. Take a typing context $\Gamma$, $s[p]:S$. If such a context is live, then, by clause $[\text{TS-Comm}]$ of Fig. 5(5), if $S$ is an external choice, then $\Gamma$ can reduce until some branch of $S$ is triggered; and by clause $[\text{TS-Comm}]$, if $S$ is an internal choice, then $\Gamma$ can reduce allowing to send each message of $S$. The clauses of liveness⁺ are stricter: they ensure that, under “fair scheduling” (details below) the interaction with $S$ will be enabled in a finite number of steps. The clauses of liveness++⁺ are even stricter, and ensure that the interaction with $S$ will be enabled within a finite number of steps, no matter how other roles are scheduled. We will give examples and more explanations shortly (Ex.5.10, Ex.5.11, Ex.5.14, Thm.5.15). But first, we explain what “under fair scheduling” means: roughly, we ensure that there is a set of roles whose interactions always cause a desired input/output to meet a corresponding output/input. This requires some sophistication, and the formalisation of the “fair traversal set” mentioned in the definition of liveness⁺ (Fig. 5(6)).

**Definition 5.5 (Fair traversal set).** Let $\mathcal{X},\mathcal{Y}$ be sets of typing contexts. We say that $\mathcal{X}$ is a fair traversal set with targets $\mathcal{Y}$ iff $\mathcal{X}$ is closed under the rules:

$$
\Gamma \in \mathcal{Y} \quad \text{TS-Target} \quad \exists s, p, q : \exists m : \Gamma \xrightarrow{s.p.q.m} \forall m : \Gamma \xrightarrow{s.p.q.m} \Gamma' \quad \text{implies} \quad \Gamma' \in \mathcal{X} \quad \text{TS-Comm}
$$

Def. 5.5 says that if a fair traversal set $\mathcal{X}$ contains a typing context $\Gamma$, then $\mathcal{X}$ also contains (part of) $\Gamma$’s reductions (inductive rule $\text{TS-Comm}$), reaching one of the target contexts in $\mathcal{Y}$ (base rule $\text{TS-Target}$). Notably, by rule $\text{TS-Comm}$, for each reduction of $\Gamma$, it is enough to choose just two roles $p, q$ who can interact (clause “$\exists m : \Gamma \xrightarrow{s.p.q.m}$”), as long as, for all interactions they can engage in, the corresponding reducend belongs to $\mathcal{X}$ (clause “$\ldots \Gamma' \in \mathcal{X}$”). Consequently, if we prove that $\mathcal{X}$ is a fair traversal set with targets $\mathcal{Y}$, then any $\Gamma \in \mathcal{X}$ is supported by an inductive derivation $\mathcal{D}$ — that, in turn, shows how we can reach some $\Gamma' \in \mathcal{Y}$ in a finite number of steps, by choosing a set of participants and following any of their possible interactions (one per instance of $\text{TS-Comm}$ in $\mathcal{D}$).

**Example 5.6.** By Def. 5.5, fair traversal sets are inductively defined: this excludes cases where target elements are reachable, but can be “infinitely delayed” by choices and recursion. E.g., let:

$$
\Gamma = s[p] ; \mu t . q \odot \{ m_1 . t , m_2 \} , \quad s[q] ; \mu t . p \& \{ m_1 , t , m_2 . r \odot m_3 \} , \quad s[r] ; q \& m_3
$$

and thus, $\Gamma \xrightarrow{s.p.q.m} s.q.r.m_3 \xrightarrow{s.p.q.m_3} \Gamma'$.

Note that $\Gamma$ is live, and $\Gamma'$ is reachable — and yet, we cannot define a fair traversal set $\mathcal{X}$ containing $\Gamma$, with a target set $\mathcal{Y} = \{ \Gamma' \}$. This is because $p, q$ can interact infinitely by exchanging $m_3$, yielding the infinite run $\Gamma \xrightarrow{s.p.q.m_1} \Gamma \xrightarrow{s.p.q.m_2} \ldots$; consequently, to support $\Gamma \in \mathcal{X}$ we would need an inductive derivation with an infinite series of instances of rule $\text{TS-Comm}$ — i.e., the derivation would be invalid.

**Example 5.7.** Fair traversal sets can be defined when elements of the target set are reachable, but can be infinitely delayed “unfair scheduling.” E.g., consider:

$$
\Gamma = s[p] ; q \odot m_1 , q' \odot m_2 , \quad s[q] ; p \& m_1 , s[q'] ; p \& m_2 , \quad s[r] ; r' \odot m_2 . t , s[r'] ; r \odot m_2 . t
$$

$\Gamma' = s[p] ; \mu t . q \odot m_1 , s[q] \odot m_1 , s[q'] \odot m_1 , s[r] ; \mu t . r \odot m_1 . t , s[r'] ; \mu t . r \odot m_1 . t$

Note that $\Gamma$ is live, and $\Gamma'$ is reachable from $\Gamma$, via the reductions $\Gamma \xrightarrow{s.r.r'.m_2} \Gamma \xrightarrow{s.r.r'.m_3} \ldots$ that never fires the communication between $p$ and $q$, and thus, never enables the interaction between $p$ and $q'$. Yet, unlike Ex.5.6, we can define a fair traversal set $\mathcal{X} = \{ \Gamma, \Gamma' \}$, with target $\mathcal{Y} = \{ \Gamma' \}$: in fact, we can build a finite derivation that supports $\Gamma \in \mathcal{X}$ by instantiating rule $\text{TS-Comm}$ twice — choosing $p, q$ for the fist reduction, and then $p, q'$ to reach the axiom $\text{TS-Target}$, ignoring the interactions between $r, r'$. 
Table 1. Verification of the multiparty protocols in Fig.4 against the properties in Fig.5. The results for protocol (3) hold for \( n \geq 1 \), while the results for protocol (4) hold for \( n \geq 2 \).

<table>
<thead>
<tr>
<th></th>
<th>consistent</th>
<th>safe</th>
<th>deadlock-free</th>
<th>live</th>
<th>live+</th>
<th>live++</th>
<th>never-terminat.</th>
<th>terminat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) OAuth2 fragment</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>(2) Rec. two-buyers</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>(3) Rec. map/reduce</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>(4) MP workers</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Ex. 5.6 and Ex. 5.7 clarify why live+ in Fig.5(6) requires the existence of a certain traversal set: this ensures that, when \( \Gamma \) has some pending input/output, then under “fair scheduling,” \( \Gamma \) can reach a target \( \Gamma_t \) where such input/output has been fired, by interacting with a matching output/input.

5.4 Relationships Between Typing Context Properties

We now study how typing context properties are related: this is formalised in Lemma 5.9 below, that also conveys the expressiveness of our new type system (Remark 5.12).

To cover classic MPST theory, we first define projected typing contexts, in Def. 5.8; note that the projections with plain and full merging correspond to claims (C1) and (C2) in §3.1, respectively.

**Definition 5.8.** We say that \( \Gamma \) is the full (resp. plain) projection of \( G \) for session \( s \), written \( \text{fproj}_{G,s}(\Gamma) \) (resp. \( \text{pproj}_{G,s}(\Gamma) \)), iff \( \Gamma = \{ s[p] : G|_{p \in \text{roles}(G)} \} \), where \( G|_p \) is the projection with full merging (resp. plain merging) in Def. 3.3.

**Lemma 5.9.** For all \( \Gamma \), the following (non-)implications hold:

1. \( \text{consistent}(\Gamma) \iff \Rightarrow \text{safe}(\Gamma) \);
2. \( \text{live}(\Gamma) \iff \Rightarrow \text{safe}(\Gamma) \);
3. \( \text{live}(\Gamma) \iff \Rightarrow \text{df}(\Gamma) \);
4. \( \text{nterm}(\Gamma) \iff \Rightarrow \text{df}(\Gamma) \);
5. \( \text{consistent}(\Gamma) \iff \Rightarrow \text{df}(\Gamma) \);
6. \( \text{consistent}(\Gamma) \land \text{df}(\Gamma) \iff \Rightarrow \text{live}(\Gamma) \);
7. \( \text{live}^{++}(\Gamma) \iff \Rightarrow \text{live}^{+}(\Gamma) \iff \Rightarrow \text{live}(\Gamma) \);
8. \( \text{term}(\Gamma) \iff \Rightarrow \text{live}^{++}(\Gamma) \);
9. **assume** \( \text{dom}(\Gamma) = \{ s \} \) (Def. 2.6). Then:
   \[ \exists G : \text{fproj}_{G,s}(\Gamma) \iff \Rightarrow \text{live}^{+}(\Gamma). \]

In the diagram, the “safe” set contains all typing contexts supported by our general type system. The red subsets are the classic MPST theory: \( G \) contains all contexts projected by some global type; its subset \( G- \) only has consistent typing contexts, i.e. the only class of global types for which classic MPST proves type safety: this class excludes our example in §1, and also all protocols in Fig.4, and more (see Ex.5.10 and Ex.5.11 below). Notably, in item (9), we prove that all projected contexts are live+: this is discussed in Remark 5.16 later.

**Example 5.10.** The protocols described in Fig.4 are verified in Table 1. We observe:

- all protocols are safe and live, but none of them is consistent: hence, they are not supported by the classic MPST theory;
- all protocols are live+, except recursive two-buyers (2): this is because it allows alice and bob to bargain forever by exchanging split/no messages, without ever involving the store (that
will keep waiting for alice to send either buy or no). This violates clause [L-\&+] of Fig.5(6), because we cannot find any traversal set whose targets trigger the store’s pending input (the issue is similar to Ex.5.6);

- two protocols are not live\(^++\): recursive two-buyers (as expected, by the point above and the contrapositive of Lemma 5.9(7)), and MP workers (4). The latter is not live\(^++\) because each triplet of workers wa\(_i\), wb\(_i\), wc\(_i\) \((i \in \{1..n\} \geq 2\) can loop independently from the others; therefore, the interaction between, e.g., two workers in triplet 1 might be delayed for an unbounded number of transitions, while triplet 2 keeps progressing. Note that this scenario arises if the roles are scheduled unfairly; otherwise, each enabled interaction \(will\) be eventually fired, and this is reflected by the fact that the MP workers protocol is live\(^+\);

- only the OAuth\(_2\) fragment (1) is terminating — while the other protocols are \(neither\) terminating, \(nor\) never-terminating: i.e., they might loop forever, but depending on the choices of one or more roles, they can reach a terminated state (where all roles have type end).

**Example 5.11.** We now provide some more small examples of multiparty protocols and their properties, complementing those discussed Ex.5.10.

\[
\Gamma_A = s[p],q; r,r \rightarrow_m, s[q]; r,r \rightarrow_m, p \rightarrow_m, s[r]; p,p \rightarrow_m q \rightarrow_m 2 \text{ is consistent (hence safe), but not live nor deadlock-free: this is because its inputs/outputs, albeit dual, occur in the wrong order.}
\]

\[
\Gamma_B = s[p],\mu t,q \rightarrow_m 1, t, s[q]; \mu t,p \rightarrow_m 1, t, s[r]; p,p \rightarrow_m 2 \text{ is consistent, deadlock-free and safe, but not live: in fact, } s[p],s[q] \text{ reduce infinitely, but } s[r] \text{ cannot fire its input (violating [L-\&+] in Fig.5).}
\]

\[
\Gamma_C = s[p]; s[q]; p \rightarrow_m (S). end \text{ with } S = \mu t,q \rightarrow_m (t). end \text{ (from [Bernardi and Hennessy 2016, Ex.1.2]) is terminating (hence live\(^++\), and safe), but not projectable from any global type, nor consistent: this is because a recursion variable } t \text{ occurs as payload in } S, \text{ which is disallowed by Def. 3.3 and Def. 3.8. Notably, } \Gamma_C \text{ types the process below (from [Bernardi and Hennessy 2016, Ex.1.2]): it creates infinitely many sessions } s' \text{ where } p \text{ and } q \text{ exchange one message } m \text{ (note that this process, although deadlock-free, does not satisfy Def. 5.5(2)).}
\]

\[
\emptyset \cdot \Gamma_C \vdash \text{ def } X(x:S,y:p \rightarrow_m (S)) = P \text{ in } X(x[p],s[q])\text{ where } P = (vs'::\Gamma_C'[S]) \left(x[q] \rightarrow_m (s'[p]).X(z,s'[q]) \right) \text{ with } \Gamma_C'[S] = s'[p]; s[q]; p \rightarrow_m (S). end
\]

**Remark 5.12.** By Lemma 5.9(1,9), our general session type system instantiated with \(\varphi = \text{fproj}_G,\&\) subsumes the classic MPST theory, and also proves subject reduction and type safety in presence of “full-merging” global type projections: this is because consistency/projectability are limited syntactic approximations of safety/liveness. Hence, the typing rule \([\Gamma-\&\rightarrow_C]\) in §3 is valid in our theory, and we can type our opening example (Ex. 4.7), and support complex protocols rejected by classic MPST, such as all those listed in Fig.4. This retroactively fixes some flawed results in literature, described in §3.1 (claim (C2)), and impacting the works listed in §8. Further, we support protocols for which no global type exists: see Ex.5.10 (case “recursive two-buyers”) and Ex.5.11 (case \(\Gamma_C\)).

### 5.5 Static Verification of Run-Time Process Properties

We now show that, by using the type-level properties in Fig.5, we can predict and constrain the run-time behaviour of processes. Roughly, the intuition is: if we have \(\Gamma \vdash P\), and some property in Fig.5 holds for \(\Gamma\), then a similar corresponding property from Def. 5.1 holds for \(P\). From this it follows that, to ensure that a closed process \((vs) P\) has a desired property from Def. 5.1, we can correspondingly instantiate \(\varphi\) in Def. 4.6, and check if the judgement “\(\emptyset \vdash (vs) \Gamma\) \(P\) with \(\varphi\)” holds.

First, we highlight that all typing context properties mentioned thus far are decidable (Thm.5.13 below) — unlike the run-time process properties in Def. 5.1. This is clear for consistency and projectability, that are syntactic and inductive; others (safety, liveness,...) are decidable because,
by Def. 2.8, typing contexts have finite-state transition systems. Consequently, by Thm. 4.11, type checking is decidable, if \( \varphi \) is instantiated with any property listed in Thm. 5.13.

**Theorem 5.13 (Decidability of \( \varphi \)).** \( \varphi(\Gamma) \) is decidable, for all \( \Gamma \), and for all \( \varphi \) such that

\[
\varphi \in \{ \text{consistent}, \text{fproj}_{G,a}, \text{pproj}_{G,a}, \text{safe}, \text{term}, \text{nterm}, \text{df}, \text{live}, \text{live}^+, \text{live}^{++} \} \quad (\text{for any } G)
\]

Now, assume \( \Gamma \vdash P \). To predict the run-time behaviour of \( P \) from \( \Gamma \), we need to overcome a complication: it might seem that if \( \Gamma \) is live (Fig. 5(5)), then \( P \) should be live, too. But this is not the case, due to a subtle interaction between the typing rule \([T\text{-Sub}]\) in Fig. 2, and the fact that supertyping does not preserve liveness: this issue (that is related to the problem of fair subtyping, studied by Padovani [2016]), is illustrated in Ex. 5.14 below. For this reason, in Thm. 5.15 we guarantee process liveness via the stronger type-level property live\(^*\): this is the payoff of fair traversal sets (Def. 5.5).

**Example 5.14.** Take \( \Gamma \) with the rec. two-buyer protocol (Fig. 4(2)): it is live (Table 1). Now, let:

\[
\Gamma' = \begin{cases}
    \{s[a]: s@query(\text{Str}), s@\text{price}(\text{Int}), \mu t.b@\text{split}(\text{Int}), b@\{\text{yes}(\text{Int}).s@\text{buy}, \text{no}.t\}\} \\
    \{s[b]: a@\text{query}(\text{Str}), a@\text{price}(\text{Int}), a@\{\text{buy}.\text{end}, \text{no}.\text{end}\}\} & (\text{as in Fig. 4(2)})
\end{cases}
\]

i.e., the types of \( a \)lice and \( b \)ob in \( \Gamma' \) are supertypes (Def. 2.5) of those in \( \Gamma \): \( a \)lice never chooses to send cancel to \( b \)ob, who in turn always answers no to all split proposals. We have \( \Gamma \leq \Gamma' \) (Def. 2.5) and \( \Gamma' \) is safe (Lemma 4.5), but not live: after sending the price, the store will wait for either buy or no from \( a \)lice, but neither message will ever be sent, while \( a \)lice and \( b \)ob loop by exchanging split/no. Consequently, a process \( P \) typed by \( \Gamma' \) can have two sub-processes implementing \( a \)lice and \( b \)ob that interact forever, while a sub-process implementing the store waits for a buy/no message, but will never receive it: hence, \( P \) is not live, as it does not satisfy Def. 5.1(2). Now, note that such \( P \) is also typed by \( \Gamma \) (via rule \([T\text{-Sub}]\) in Fig. 2): i.e., a live typing context can type a non-live process.

**Theorem 5.15.** Assume \( \emptyset \cdot \Gamma \vdash P \), with \( \Gamma \) safe, \( P \equiv \bigcup_{p \in \rho} P_p \), each \( P_p \) having guarded definitions and either being \( \emptyset \) (up-to \( \equiv \)), or only playing role \( p \) in \( s \). Then, (1) \( \text{df}(\Gamma) \) implies that \( P \) is deadlock-free; (2) \( \text{term}(\Gamma) \) implies that \( P \) is terminating; (3) \( \text{nterm}(\Gamma) \) implies that \( P \) is never-terminating; (4) \( \text{live}^+(\Gamma) \) implies that \( P \) is live; and (5) \( \text{live}^{++}(\Gamma) \) implies that \( P \) is strongly live.

**Proof.** The results follow by Thm. 5.4 (session fidelity). For (4) we also use the fact that, if \( \text{live}^+(\Gamma) \) and \( \Gamma \leq \Gamma' \), then \( \text{live}^+(\Gamma') \).

**Remark 5.16.** With Lemma 5.9(9) and Thm. 5.15(4), we uncover that global types / projections (Fig. 3) are ways to produce live\(^*\) typing contexts, and ensure that processes are live. Since Thm. 5.15 does not need the technicalities of Fig. 3, our theory and results are more general than classic MPST. And importantly, the premises of all cases of Thm. 5.15 are decidable (by Thm. 5.13 and Thm. 4.11).

### 6 VERIFYING TYPE-LEVEL PROPERTIES VIA MODEL CHECKING

Our new MPST theory (§ 4) is parametric on a general property \( \varphi \), that is not constrained by syntactic duality/consistency. In this section, we leverage this distinguishing feature to integrate type checking and model checking, in two steps: (1) we show how to express \( \varphi \) as a modal \( \mu \)-calculus formula, and (2) we use a model checker (through the paper’s companion artifact) to verify whether the transitions of \( \Gamma \) satisfy the \( \mu \)-calculus version of \( \varphi \). This provides a practical method to verify whether \( \varphi(\Gamma) \) holds — e.g., in rule \([T\text{Gen-v}]\) (Def. 4.6), and in Thm. 5.15.

We focus on a fragment of the \( \mu \)-calculus with data, adopting a formulation based on [Groote and Mousavi 2014, §6.5]. Let \( \alpha \) range over the labels in Def. 2.8 — i.e., \( \alpha \) can have the form \( s.p \& q.m(S) \) for input, or \( s.p@q.m(S) \) for output, or \( s.p.q.m \) for communication. Then, \( \mu \)-calculus formulas are defined as follows, where \( d \) ("data") ranges over sessions, roles, message labels, and session types:
A formula $\phi$ accepts or rejects a typing context $\Gamma$ depending on the sequences of actions that $\Gamma$ can fire along its transitions. A formula can be either: true/false ($\top/\bot$), i.e., accept any/no typing context; box modality $[\alpha]\phi$ ("for all transitions with label $\alpha$, the reached typing context must satisfy $\phi$"); diamond modality $\langle \alpha \rangle \phi$ ("for some transition with label $\alpha$, the reached typing context satisfies $\phi$"); implication $\Rightarrow$; least/greatest fixed point $\mu Z. \phi/\nu Z. \phi$, allowing to iterate $\phi$ for a finite/infinite number of times; a variable $Z$, for iteration; and universal/existential quantification $\forall \phi/\exists \phi$.

When a typing context $\Gamma$ satisfies a formula $\phi$, we write $\Gamma \models \phi$. 

**Example 6.1.** The $\mu$-calculus formula $\phi = \exists s. \exists p. \exists q. \exists m. (s.p@q.m(S)) \top$ says: "accept a typing context if, for some session $s$, roles $p$ and $q$, message label $m$, and type $S$, it can perform an output action $s.p@q.m(S)$" — and after such a transition, the reached typing context is always accepted, by $\top$. Therefore, if we take the typing context $\Gamma = s[r]. r'@msg(\text{Str}). \text{end}$, then we have $\Gamma \models s[r]. r'@msg(\text{Str}) \top$ (by Def. 2.8), which means that $\Gamma$ satisfies $\phi$ — in symbols, $\Gamma \models \phi$. Moreover, $\Gamma$ satisfies the formula $\forall s. \forall p. \forall q. \forall m. (s.p@q.m) \bot$, that holds when no communication is possible, for any role: in fact, the formula says that any communication would reach a context rejected by $\bot$.

Instead, if we take the formula $\phi' = \exists s. \exists p. \exists q. \exists m. (s.p@q.m) \top$, then $\Gamma$ above does not satisfy $\phi'$, because it requires a communication transition to be enabled. However, if we extend $\Gamma$ as $\Gamma' = \Gamma, s[r'] . r'@msg(\text{Str}). \text{end}$, then we have both $\Gamma' \models \phi'$ and $\Gamma' \models \phi'$ — and thus, $\Gamma' \models \phi \land \phi'$.

**Example 6.2 (Formulas in Fig. 5).** We now describe the $\mu$-calculus formulas in Fig. 5:

- **safety** (1) checks that, if an output $m$ and an input $m'$ are enabled between two roles $p$ and $q$, then they can communicate via $m$ (i.e., by Def. 2.8, the output message $m$ must be supported by the recipient). This must hold for any context reachable via communication transitions: this is enforced by the greatest fixed point $\nu Z. \ldots$ and the clause $\ldots \land [s.p@q.m]Z$;
- **deadlock-freedom** (2) checks whether communication is possible; if not ($\forall \ldots [s.p@q.m] \bot$), that holds only when no roles can interact, cf. Ex. 6.1), then ($\Rightarrow$) there must be no input nor output transitions enabled — i.e., all typing context entries must be $\text{end}$. This must hold for any context reachable via communications: it is enforced by $\nu Z. \ldots$ and $\ldots \land \forall \ldots [s.p@q.m]Z$;
- **termination** (3) is similar to deadlock-freedom, but uses a least fixed point $\mu Z. \ldots$: hence, the clause $\ldots \land \forall \ldots [s.p@q.m]Z$ can only iterate for a finite number of times, and then no communications, nor inputs, nor outputs must be enabled — i.e., all context entries are $\text{end}$;
- **never-termination** (4) checks that in any context reachable via communication transitions ($\nu Z. \ldots$ and $\ldots \land \forall \ldots [s.p@q.m]Z$), some further communication is possible ($\exists \ldots (s.p@q.m) \top$);
- **liveness** (5) checks that, if an input or output between two roles $p$ and $q$ is enabled, then ($\Rightarrow$) a corresponding communication can be fired, after a finite sequence of communications among any role. The sequence is built with a least fixed point $\mu Z'. \ldots$, that can iterate on the clause $\ldots \lor \exists \ldots (s.p'@q'.m') Z'$ for a finite number of times. The top-level greatest fixed point $\nu Z. \ldots$ repeats the check for all contexts reachable via communication (clause $\ldots \lor \forall \ldots [s.p@q.m]Z$);
- **liveness** (6) is similar to liveness, but the nested fixed points $\mu Z' \ldots$ build finite sequences of communications by picking a pair of roles $p', q'$ at each step, and following all their interactions, until a communication between $p, q$ is enabled. This corresponds to building the fair traversal set (Def. 5.5) required by the left-side definition of live$^+$ in Fig. 5;
- **liveness** (7) is also similar to liveness, but the nested fixed points $\mu Z' \ldots$ build finite sequences by following any communication between any pair of roles, until a communication between $p, q$ is enabled. This ensures that, along any execution path, after a finite number of steps, $p$ and $q$ will be able to interact, as in the left-side definition of live$^{++}$ in Fig. 5.
where Types are extended to model pending messages; e.g., the processes in (10) are typed by, respectively:

\[
\text{(4) are instantiated with } n \pm 7 \text{ ASYNCHRONOUS MULTIPARTY SESSION (q to of §2.1). The first reduction enqueues the pending message In the topmost process, s}
\]

Our

In its original formulation [Bettini et al. 2008; Honda et al. 2008], the MPST is a powerful typed calculus that allows for the specification of distributed systems. However, its original formulation had some limitations, particularly in terms of its ability to verify properties of distributed systems. Hence, mCRL2 can verify our formulas in Fig. 5 without exploring the whole state space of \(\Gamma\), as shown in Table 2. In all instances, the verification takes around one second. Notably, this also holds for the multiparty workers protocol (4), although it has 12000× more states than the OAuth2 fragment (1). This state space explosion is due to the interleaving of multiple parallel components — but still, its impact on verification time is minimal: in fact, the properties in Fig. 5 only follow the communication transitions of a typing context \(\Gamma\), whereas the input and output transitions of \(\Gamma\) are checked for their presence/absence, but not followed to their destination state. Hence, mCRL2 can verify our formulas in Fig. 5 without exploring the whole state space of \(\Gamma\).

### 7 ASYNCHRONOUS MULTIPARTY SESSION \(\pi\)-CALCULUS

In its original formulation [Bettini et al. 2008; Honda et al. 2008], the MPST \(\pi\)-calculus has asynchronous buffered semantics, to model typical "real-world" distributed message-passing programs. Our new theory extends to asynchrony, overcoming challenges due to queue handling and decidability.

**NOTE:** this section is a summary of the results that are discussed, in full detail, in §C–§G. In asynchronous MPST, we give an intuition of the asynchrony calculus with an example:

\[
s[p][q]@m(s'[r]).P \mid s[q][p]\sum m(x).Q \mid s \triangleright \epsilon \\
\rightarrow P \mid s[q][p]\sum m(x).Q \mid s \triangleright (p,q,m(s'[r])).\epsilon \rightarrow P \mid Q\{s'[r]/x\} \mid s \triangleright \epsilon \tag{10}
\]

In the topmost process, \(s \triangleright \epsilon\) is the (empty) message queue of session \(s\). As \(s\) is not present in the calculus of §2.1. The first reduction enqueues the pending message \((p,q,m(s'[r]))\), meaning that \(p\) has sent to \(q\) a message with label \(m\) and payload \(s'[r]\). With the second reduction, the message is received.

The classic async MPST typing judgement has the following form:

\[
\Theta \cdot \Gamma + s \Rightarrow P \tag{11}
\]

where \(s\) is the set of sessions whose queue occurs in \(P\) (e.g., to type (10) above, we let \(s = \{s\}\)). Types are extended to model pending messages; e.g., the processes in (10) are typed by, respectively:

\[
\Gamma = s[p];q@m(S').S, s[q];p&m(S').T, s'[r]:S' \\
\Gamma' = s[p];(q!m(S')\cdot\epsilon;S), s[q];p&m(S').T, s'[r]:S' \\
\Gamma'' = s[p];S, s[q];T, s'[r]:S' \tag{12}
\]

Note that \(\Gamma\) above is a typing context similar to Def. 2.6. Instead, in \(\Gamma'\) the type of \(s[p]\) is a pair \((M;S)\), where \(M = q!m(S')\cdot\epsilon\) is a message queue type (abstracting the pending messages sent through
s[p]), followed by the continuation type S. In Γ’, the topmost queued message type matches the branching type of s[q]: their interaction leads to Γ’’, with a reduction similar to Def. 2.8.

The classic async MPST theory has all the issues described in §3 — but the presence of message queues makes its subject reduction statement more complicated [Coppo et al. 2015a, Lemma 1]:

\[
\text{If } \Theta \cdot \Gamma \vdash_S P \text{ and } \exists \Gamma_0 \text{ such that } \Gamma, \Gamma_0 \text{ consistent and } P \rightarrow P', \\
\text{then } \exists \Gamma', \Gamma'' \text{ consistent: } \Gamma, \Gamma_0 \rightarrow^* \Gamma', \Gamma_0' \rightarrow^* \Gamma'' \text{ and } \Theta \cdot \Gamma' \vdash_S P'.
\]

**General Asynchronous MPST.** We extend our new theory in §4 to asynchronous MPST, and prove a simpler and more general subject reduction statement: Thm. 7.1. To achieve it, we develop async typing rules based on an async safety property \(\phi\), with a more sophisticated async typing context reduction \(\rightarrow_S\), where S is a set of sessions, as in (11); e.g., in (12) we have \(\Gamma \rightarrow^*_S \Gamma'' \rightarrow^*_S \Gamma''\).

**Theorem 7.1 (Async Subject Reduction).** Assume \(\Theta \cdot \Gamma \vdash_S P\) with \(\Gamma\) asynchronously safe. Then, \(P \rightarrow P'\) implies \(\exists \Gamma'\) asynchronously safe such that \(\Gamma \rightarrow_S \Gamma'\) and \(\Theta \cdot \Gamma' \vdash_S P'\).

(Un-)Decidability of Type Checking. A result akin to Thm. 4.11 holds for async MPST.

**Theorem 7.2.** If \(\phi\) is decidable, then “\(\Theta \cdot \Gamma \vdash_S P'\) with \(\phi\)” is decidable.

However, under asynchrony we do not have a decidability result for \(\phi\) as general as Thm. 5.13. On the contrary, async safety and most other properties are undecidable: the pairing of a session type with a message queue (cf. \(\Gamma\) in (12)) corresponds to a Communicating Finite-State Machine (CFSM) [Brand and Zafiropulo 1983], and makes typing contexts Turing-powerful [Bartoletti et al. 2016, Thm. 2.5]. Still, we obtain decidable instances of \(\phi\) through various sound approximations:

**M1** consistency is decidable, and implies asynchronous safety;

**M2** via the session type / CFSM correspondence established in [Deniélou and Yoshida 2013], we show that if \(\Gamma\) is synchronously live (Fig.5(5), decidable by Thm. 5.13), then \(\Gamma\) is also asynchronously live; we extend the result to live+ (Fig.5(6)); and by Lemma 5.9(9), this means that any \(\Gamma\) projected from a global type is asynchronously live+;

**M3** given \(n \geq 1\), we can decide if \(\Gamma\) enqueues at most \(n\) messages; if so, \(\Gamma\) is finite-state, hence async safety/liveness are decidable. For example, take \(\Gamma = s[p] \cdot q \oplus q. m_1. q \& m_2. s[q] \cdot p \oplus m_2. p \& m_1\): it is deadlocked under synchronous semantics, and not projectable from any global type — but under asynchrony, the top-level outputs of \(p\) and \(q\) can be both enqueued, and then received; hence, we can decide that \(\Gamma\) enqueues at most 2 messages, and is asynchronously live.

**Remark 7.3.** By instantiating \(\phi\) in Thm. 7.2 with one of the methods above, we obtain an expressive decidable fragment of our new asynchronous MPST theory: **M1** subsumes classic async MPST; **M2** covers all live typing contexts, albeit non-consistent: e.g., it covers all cases in Fig. 4, and all global types (by Lemma 5.9(9)); **M3** covers more typing contexts that are not projectable from global types.

## 8 Conclusion, Related and Future Work

We have presented a new theory of multiparty sessions types, with novel foundations that do not depend on duality/consistency, nor global types, nor projections. Our new theory subsumes classic MPST, also fixing subject reduction flaws in previous work (Remark 5.16). Moreover, our new type system is modular and reusable: by fine-tuning its parameter \(\phi\), we ensure that type-checking is decidable, and that processes are safe, deadlock-free, and live. A summary of the main results:

- **R1** our type safety results (Thm. 4.8, Cor. 4.9) are much more general than classic MPST;
(R2) if we instantiate $\phi$ with projection/consistency, or any property in Fig. 5, then the type checking judgement $\Theta \cdot \Gamma \vdash P$ with $\phi$ is decidable. This follows from Thm. 4.11 and Thm. 5.13;

(R3) by suitably choosing $\phi$ in (R2) above, we can statically guarantee that $P$ "inherits" $\phi$, and has certain desired run-time properties. This is formalised in Thm. 5.15;

(R4) we can implement $\phi$ in (R2)/(R3) above as a syntactic check (Remark 5.12), or as a $\mu$-calculus formula (Fig. 5). In the latter case, we can verify whether $\Gamma$ satisfies $\phi$ via model checking — e.g., using mCRL2, through the paper’s companion artifact (mpstk). This is shown in §6;

(R5) our new theory extends to asynchronous communication, as illustrated in §7.

8.1 Classic Multiparty Session Types (MPST)

The classic MPST framework, and its notions of global types and projections, were introduced by Honda et al. [2008], with linearity conditions to check the well-formedness of global types, and ensure projectability of local types. Later, Bettini et al. [2008] proposed a simplified MPST system adopted by most works, including ours.

We now classify some related works w.r.t. their use of projection/consistency:

<table>
<thead>
<tr>
<th>papers</th>
<th>projection</th>
<th>consistency</th>
<th>subj. red.</th>
<th>claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Bettini et al. [2008]; Carbone et al. [2016, 2015]; Coppo et al. [2015a]; Honda et al. [2008, 2016]</td>
<td>≤ plain</td>
<td>yes</td>
<td>correct</td>
<td>(C1)</td>
</tr>
<tr>
<td>Caires and Perea [2016]; Chen [2013]; Denielou et al. [2012]; Torino and Yoshida [2016]</td>
<td>≥ full</td>
<td>no</td>
<td>flawed</td>
<td>(C2)</td>
</tr>
<tr>
<td>(c) Scalas et al. [2017a]; Toninho and Yoshida [2017]</td>
<td>full</td>
<td>yes (required)</td>
<td>correct</td>
<td>(C1)</td>
</tr>
</tbody>
</table>

Row (a) lists works using plain (or stricter) global type projection (Def. 3.3), guaranteeing consistency. As shown in §5.4, our theory captures plain projection / consistency by setting its parameter $\phi$ as $\phi = pproj_{G,s}$, $\phi = \text{consistent}$; however, this excludes many valid protocols, as per claim (C1) — e.g., all our examples in Fig. 4.

Row (b) lists works using full (or more flexible) global type projection, originally introduced in Yoshida et al. [2010] to support more protocols. Such works overlook the consistency requirement; and in §3, we reveal that classic MPST subject reduction proofs relying on full projection (without consistency) are flawed, as per claim (C2). To "fix" these works within the classic MPST theory, we must require consistency, as done by works in row (c) — but this restricts typability, thus falling back into claim (C1). Instead, by Remark 5.12, our new MPST theory supports full projections with $\phi = fproj_{G,s}$, thus subsuming classic MPST and fixing flaws, without losing expressiveness.

8.2 Non-Classic Multiparty Session Types

To the best of our knowledge, there are two MPST works (mentioned in Remark 3.1) that are not based on classic projection+consistency (Fig. 3) — but have other limitations, that we surmount.

The first work is by Dezani-Ciancaglini et al. [2015] (with a more recent journal version by Ghilezan et al. [2018]): it presents a single-session type system, with first-order session types (i.e., without channel-passing); it is rooted on global types and their projections, but does not require consistency. The resulting subject reduction proof strategy is quite complex, as it requires to reason on global types and their semantics (see the proof of subject reduction in Ghilezan et al. [2018]). Such a type system is subsumed by letting $\phi = fproj_{G,s}$ in our Def. 4.6; in addition, our work also supports higher-order types, multiple interleaved sessions, and protocols for which no global type exists (see Table 1(2), and Ex. 5.11, case $\Gamma_c$).

The second non-classic MPST work is by Scalas and Yoshida [2018]: it was our first attempt (and, to the best of our knowledge, the first work in general) to directly address the limitations of
consistency (claim (C1)), and propose a behavioural theory of MPST, not based on global types and projections. Unfortunately, we could not build upon that work, due to its intrinsic limitations:

(1) a major goal of this paper is subsuming and fixing classic MPST (cf. claim (C2) in §1, and §3). However, the theory of Scalas and Yoshida [2018] cannot achieve this goal: it has different (and more complicated) typing rules that require typing context liveness, and do not support consistency. Our new theory, instead, supports both consistency and liveness, as instances of $\varphi$ (Lemma 5.9, Remark 5.12);

(2) from Scalas and Yoshida [2018], we reuse the definition of typing context liveness (Fig. 5(5)) — but we show that it is insufficient to guarantee process liveness (Def. 5.1, Ex.5.14). Hence, we develop the stronger properties live$^+$/live$^{++}$ (Fig. 5(6,7)), to obtain the results on run-time process behaviour in Thm. 5.15. Such results are absent in Scalas and Yoshida [2018];

(3) the branching/selection typing rules of Scalas and Yoshida [2018] (Fig. 3) directly inspect typing context reductions. This peculiarity is not problematic under synchronous semantics (where typing contexts have finite-state transition systems), and in some cases, it enables flexible typing judgements that cannot be obtained in classic MPST [Scalas and Yoshida 2018, Ex. 5.5]. The drawback is that, when extended to asynchronous semantics, typing contexts become Turing-powerful (§7), and typing rules that inspect their reductions become inherently undecidable; consequently, the theory of Scalas and Yoshida [2018] does not subsume classic works on asynchronous MPST, and cannot achieve this goal without a major overhaul. Instead, our typing rules do not inspect typing context reductions, but only check whether the parametric property $\varphi$ holds: hence, type checking is decidable whenever $\varphi$ is decidable (Thm. 7.2), and this allows us to subsume classic asynchronous MPST (Remark 7.3).

By instantiating $\varphi =$ live in Def. 4.6, this paper largely subsumes Scalas and Yoshida [2018]’s work — minus some corner cases based on the inspection of typing context reductions (cf. item (3) above).

8.3 Binary Sessions Without Duality

Our work yields a generalised theory of binary sessions not based on classic duality (Def. 3.5), subsuming classic papers based on [Honda et al. 1998]. If we take a binary session typing context $\Gamma = [p : S, q : T]$, our Lemma 5.9 becomes:

$$\exists G: fproj_{G,f}(\Gamma) \iff \text{consistent}(\Gamma) \iff \text{live}^{++}(\Gamma) \iff (\text{safe}(\Gamma) \text{ and df}(\Gamma))$$  \hspace{1cm} (13)

Here, the leftmost “$\iff$” is due to supertyping: e.g., if we take the global type $G = p \to q : \{m, m\}'$, it projects the typing context $\Gamma = [p : q \oplus \{m, m\}', q : p \& \{m, m\}']$, that is consistent and live$^{++}$ (hence safe); however, if we replace $p$’s entry with the supertype $p \oplus m$, the resulting context is still live$^{++}$ and consistent, but not projectable from any global type. The other “$\iff$” in (13) is due to non-tail-recursive types like $\mu t. q \oplus m(t). \text{end}$: they have no dual in classic binary session types (since $t$ is a forbidden payload): thus, they yield non-consistent typing contexts, and processes like $P$ in Ex.5.11 (case $G_C$) cannot be typed. This limitation has been addressed by several authors, extending duality with various pitfalls (see e.g. [Bernardi and Hennessy 2016, §5.3]): for a survey, and a logic-based solution, see [Lindley and Morris 2016, §3.2]. By not using duality, our theory eschews these issues.

8.4 Type Systems for the $\pi$-Calculus

Many type systems have been proposed for the $\pi$-calculus, also influencing MPST: see survey in [Hüttel et al. 2016]. Our new MPST theory is a case of behavioural type system: it treats types as simple processes that reduce and evolve along a typed computation; and since types are simpler than programs, they can be analysed with simpler methods (e.g., finite model checking via our parameter $\varphi$, cf. §6). As stated in §4, the design of our new MPST theory is inspired by Igarashi and
Kobayashi [2004]'s Generic Type System (GTS) for the \( \pi \)-calculus: i.e., we define a type system that is parametric on a property \( \varphi \), and we prove type safety under the weakest \( \varphi \); then, we fine-tune \( \varphi \) to statically verify stronger properties of processes, like deadlock-freedom and liveness (§5). Besides this general analogy, our treatment is wholly different: we carefully reuse fundamental MPST definitions (§2.1) and develop new and more general results (§4, §5) to ensure our new theory fully subsumes the classic one; moreover, for async MPST we devise a new treatment of queue types, obtaining a new, more general subject reduction result (Thm.7.1).

As an alternative, we might have tried to encode MPST in the GTS, and develop our new results from there. However, this appears unfeasible. Gay et al. [2014] tried the approach for binary sessions, obtaining drawbacks in terms of complication and loss of abstraction (see "Assessment" in Gay et al. [2014]): such drawbacks would be greatly amplified for multiparty sessions. Moreover, [Igarashi and Kobayashi 2004, §4.2, §5] study process/type correspondence using a temporal logic without fixed points, with limited support for recursion: their logic would not allow, e.g., to model our variants of liveness (Fig.5) and address the interplay between liveness, subtyping, and recursion (Ex.5.14, Thm.5.15). Further, the encoding approach would not work for async MPST: the types of Igarashi and Kobayashi [2004] lack message queues, and are akin to CCS without restriction, with decidable reachability [He 2011, p. 374]; hence, they cannot encode the Turing-powerful typing contexts of async MPST (§7), whose reachability is undecidable.

8.5 Choreographies and Communicating Finite-State Machines (CFSMs)

Various works model and verify multiparty protocols, a.k.a. choreographies, via automata-theoretic methods, by representing each party as a CFSM [Brand and Zafiropulo 1983]. The safety their interactions (that is generally undecidable) is verified with two main approaches: (a) assume the decidability of a synchronisability property [Basu and Bultan 2011, 2016; Basu et al. 2012], and then check temporal properties of CFSMs via model checking; (b) check decidable synchronous execution conditions on CFSMs, and prove that they ensure safe asynchronous executions [Bocchi et al. 2015; Deniélou and Yoshida 2013; Lange et al. 2015]. Both methods can help extending our new MPST theory: since we essentially treat async typing contexts as systems of CFSMs (§7), new decidable results on CFSM safety can yield new decidable instances of our type system (Thm.7.2). Unfortunately, synchronisability has been recently proven undecidable by Finkel and Lozes [2017]: i.e., method (a) above might be unusable — hence, we adopt method (b) (cf. (M2) in §7). Unlike this paper, the above CFSM works do not study type systems, nor properties of typed processes.

8.6 Future Work

We kept our typing rules close to classic MPST, to easily combine our results with existing works. E.g., we plan to integrate our work with Coppo et al. [2015b], that studies MPST deadlock-freedom in presence of multiple interleaved sessions: our generalised typing rules can be a drop-in replacement for the classic rules used by Coppo et al. [2015b], and this integration would combine their global deadlock-freedom checks, with our improved type safety results for individual sessions. We also plan to extend the calculus (e.g., with polymorphism [Caires and Pérez 2016; Goto et al. 2016]), and expand the properties/formulas studied in Fig.5 and Thm.5.15. We will investigate the logical foundations of our new MPST theory, aiming at results that generalise those by Carbone et al. [2016, 2015], which are focused on limited global types, projections, and consistency.

Another interesting research topic is the completeness of safety (Def. 4.1), i.e., studying whether the inverse implication w.r.t. Thm.4.8/Cor.4.9 holds. This corresponds to the following conjecture:

\[ \text{Take any } \Gamma. \text{ If } \forall P, P': \Gamma \vdash P \text{ and } P \rightarrow^* P' \text{ implies that } P' \text{ has no error, then } \text{safe}(\Gamma). \]
We will investigate whether this conjecture holds — and if not, what other completeness results are achievable. Since session subtyping is central for defining safety (via clause \([S \rightarrow C]\) in Def. 4.1, and \([\Gamma-\text{Comm}]\) in Def. 2.8), we will leverage Chen et al. [2017]'s work on the completeness of subtyping.

We will also study how to implement our new MPST theory. A basis is the work by Scalas et al. [2017a,b], that embeds classic MPST in Scala, through a linear \(\pi\)-calculus encoding based on consistency; however, since we do not require consistency, the work by Scalas et al. [2017a,b] only covers a fragment of our new theory. Using the \(\mu\)-calculus formulas illustrated in §6, a new implementation can verify typing context properties by offloading them to a model checker.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful remarks. Thanks to Francisco Ferreira, Sung-Shik Jongmans, and Julien Lange for their comments, and to Simon Castellan for testing the companion artifact. This work was partially supported by EPSRC (projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1), and by the EU COST Action CA15123 (“EUTypes”).

REFERENCES


Luís Caires and Jorge A. Pérez. 2016. Multiparty Session Types Within a Canonical Binary Theory, and Beyond. In FORTE. https://doi.org/10.1007/978-3-319-39570-8_6

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. MSCS 26, 3 (2016).


Alain Finkel and Etienne Lozes. 2017. Synchronizability of Communicating Finite State Machines is not Decidable. In ICALP. https://doi.org/10.4230/LIPIcs.ICALP.2017.122


Simon J. Gay, Nils Gesbert, and António Ravara. 2014. Session Types as Generic Process Types. In EXPRESS/SOS. https://doi.org/10.4204/EPTCS.160.9


Julien Lange, Emilio Tuosto, and Nobuko Yoshida. 2015. From Communicating Machines to Graphical Choreographies. In POPL. https://doi.org/10.1145/2676726.2676964


## Appendices — Part 1

### Additional definitions, and asynchronous MPST

<table>
<thead>
<tr>
<th></th>
<th>Synchronous</th>
<th>Asynchronous (§7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiparty session π-calculus</td>
<td>§2.1</td>
<td>§C</td>
</tr>
<tr>
<td>Multiparty session types</td>
<td>§2.2</td>
<td>§D</td>
</tr>
<tr>
<td><strong>Issues of classic MPST</strong></td>
<td>§3</td>
<td>§E</td>
</tr>
<tr>
<td><strong>A new, general MPST theory</strong></td>
<td>§4</td>
<td>§F</td>
</tr>
<tr>
<td>• Typing context safety invariant</td>
<td>Def. 4.1</td>
<td>Def. F.2</td>
</tr>
<tr>
<td>• Subject reduction</td>
<td>Thm.4.8</td>
<td>Thm.7.1 / Thm.F.6</td>
</tr>
<tr>
<td>• Decidability of type checking</td>
<td>Thm.4.11</td>
<td>Thm.7.2</td>
</tr>
<tr>
<td><strong>Verifying process behaviours using types</strong></td>
<td>§5</td>
<td>§G</td>
</tr>
<tr>
<td>• Session fidelity</td>
<td>Thm.5.4</td>
<td>Thm.F.8</td>
</tr>
<tr>
<td>• Typing context properties</td>
<td>Fig.5, Def. 5.8</td>
<td>Def. G.2</td>
</tr>
<tr>
<td>• Static verification of process properties</td>
<td>Thm.5.15</td>
<td>Thm.G.9</td>
</tr>
<tr>
<td>• (Un-)Decidability of typing ctx. properties</td>
<td>Thm.5.13</td>
<td>Thm.G.5, Thm.G.6</td>
</tr>
<tr>
<td><strong>Model checking of type-level properties</strong></td>
<td>§6</td>
<td>Remark G.10</td>
</tr>
<tr>
<td><strong>Related and Future Work</strong></td>
<td>§8 and §H</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Contents and contributions of this paper. The contents and main contributions in the “Asynchronous” column are summarised in §7. Proofs are available in the second part of the appendices (§I–§N).
which completes Fig. 1. In the rules above, \(fpv\). This section discusses some intermediate results leading to Thm.5.4 (session fidelity). For the full
\[ \pi \]

The standard congruence delation of the MPST \(\pi\)-calculus, mentioned in Fig.1, is formalised in Fig.6. The up-to congruence reduction rule \([R\equiv]\), which was omitted in Fig.1, says that reduction is closed under \(\equiv\).

\section{Multiparty Session \(\pi\)-Calculus}

The standard congruence delation of the MPST \(\pi\)-calculus, mentioned in Fig.1, is formalised in Fig.6. The up-to congruence reduction rule \([R\equiv]\), which was omitted in Fig.1, says that reduction is closed under \(\equiv\).

\section{Session Fidelity}

This section discusses some intermediate results leading to Thm.5.4 (session fidelity). For the full proof details, see §I.

The MPST typing system enjoys the fundamental Lemmas B.1 to B.3 below: they hold in most typing systems, and will be necessary for session fidelity (§5.2).

\begin{lemma}[Substitution] Assume \(\Theta \cdot \Gamma, x:S \vdash P\) and \(\Gamma' \vdash s[p]:S\), with \(\Gamma, \Gamma'\) defined. Then, \(\Theta \cdot \Gamma, \Gamma' \vdash P[s[p/x]\}.
\end{lemma}

\begin{proof}
Minor adaptation of [Coppo et al. 2015a, Lemma 5].
\end{proof}

\begin{lemma}[Subject Congruence] Assume \(\Theta \cdot \Gamma \vdash P\) and \(P \equiv P'\). Then, \(\Theta \cdot \Gamma \vdash P'\).
\end{lemma}

\begin{proof}
By examining the cases where \(P \equiv P'\) holds, and by inversion of the typing judgements \(\Theta \cdot \Gamma \vdash P\) and \(\Theta \cdot \Gamma \vdash P'\) (Lemma I.4).
\end{proof}

\begin{lemma}[Narrowing] If \(\Theta \cdot \Gamma \vdash P\) and \(\Gamma' \subseteq \Gamma\), then \(\Theta \cdot \Gamma' \vdash P\).
\end{lemma}

With Def. 5.3 and Lemmas B.1 and B.2, we can show how typing contexts determine process shapes. Thm.B.4 below addresses the typical application scenario of MPST, i.e., an ensemble of programs \(P_p\) that interact on a multiparty session \(s\), each one playing a distinct role \(p\). The crucial parts are items (1) and (2): if the type of the channel used by \(P_p\) requires to select (resp. branch), then \(P_p\) will be ready to perform the corresponding operation, possibly after calling some \(X\). Note that this crucially relies on item 1 of Def. 5.3. In fact, by rule \([T\text{-def}]\) (Fig.2), an unguarded definition like \(X(x:S) = X(x)\) can be typed with any \(S\), even when \(S\) is an internal/external choice requiring to use \(x\) for selection/branching. This possibility would refute items (1)(b) and (2)(b) of Thm.B.4 — but item 1 of Def. 5.3 solves the issue: it forces unused process parameters to be end-typed.

\begin{theorem}[Session Inversion] Assume \(\Theta \cdot \Gamma \vdash \bigcup_{p \in I} P_p\) with each \(P_p\) either being \(\Theta\) (up-to \(\equiv\)), or only playing role \(p\) in \(s\). Then, \(\Gamma \equiv \Gamma_0, \{s[p]: S_p\}_{p \in I'}\) (for some \(I'\)) with \(\text{end}(\Gamma_0)\). Moreover, \(\forall p \in I'\);
We require that in well-formed processes, we now address asynchronous MPST will require us to address several challenges:

1. for queues in Fig. 7: the first is for garbage collection; the second reorders messages with different sender/recipient. E.g.:
   - an unsupported message. The semantics is defined up-to the congruence error rule pending message if its sender, recipient, and label match the receiving process. The set of senders in rules in Fig. 7. The set of 

\[ \text{\[R-Err\]} \]

\[ \text{\[R-AErr\]} \]

\[ \text{\[R-AOut\]} \]

\[ \text{\[R-AIn\]} \]

Further, (4) \( \forall p \in I \setminus I' : P_p \equiv 0. \)

C ASYNCHRONOUS MULTIPARTY SESSION \( \pi \)-CALCULUS

We now address asynchronous MPST, as in the original MPST papers [Bettini et al. 2008; Honda et al. 2008] and in most successive works. Asynchronous MPST provide a more faithful model of real-world distributed applications, that usually employ buffered message-passing (e.g., via the TCP protocol); moreover, we will see that our new async MPST theory (unlike the classic one) can handle protocols whose correctness actually depends on message buffering (Ex. G.4(4),(5)). However, asynchrony will require us to address several challenges:

1. the classic async MPST theory has additional complications (§E);
2. to eschew such complications, and successfully extend our approach to asynchrony, we will develop a new proof strategy for subject reduction (§F);
3. async typing context properties are generally undecidable; still, we will achieve decidable type checking by leveraging results from communicating automata (§G).

In this section, we start developing the asynchronous theory by adding non-blocking send operations and message queues to the \( \pi \)-calculus of §2.1. From now on, we overload the notation of §2.1, and focus on the differences between synchronous/asynchronous definitions and results.

Definition C.1. Async MPST processes have the syntax in Def. 2.1, plus session queues:

\[ P, Q ::= \ldots | s \triangleright \sigma \quad \text{where } \sigma \text{ is a message queue: } \sigma ::= (p, q, m(s'[r])) \cdot \sigma \mid \epsilon \]

We require that in well-formed processes, each session has a queue: if \( P = (vs) Q \), then \( Q \equiv (vs') (Q' \mid s \triangleright \sigma) \).

The semantics of async processes is induced by the rules in Fig. 1, but (1) we replace \([R-Comm]\) and \([R-Err]\) by \([R-AOut]\), \([R-Abs]\) and \([R-AErr]\) in Fig. 7, and (2) the congruence \( \equiv \) is extended with the rules in Fig. 7. The set of message senders in \( \sigma \), or senders(\( \sigma \)), is:

senders((\( p, q, m(s'[r]) \)) \cdot \sigma') = \{p\} \cup \text{senders}(\sigma') \quad \text{senders}(\epsilon) = \emptyset

In Fig. 7, the output rule \([R-AOut]\) enqueues a pending message − i.e., a triple with the message sender role, the intended recipient, and the message itself. The input rule \([R-Abs]\) dequeues a pending message if its sender, recipient, and label match the receiving process. The error rule \([R-AErr]\) fires if a process with role \( p \) is waiting for a message \( m_i (i \in I) \) from \( q \), but the queue head is an unsupported message. The semantics is defined up-to the congruence \( \equiv \) in Fig. 6, plus the rules for queues in Fig. 7: the first is for garbage collection; the second reorder messages with different sender/recipient. E.g.:

\[ s[p][q] \sum m_2(x).P \mid s \triangleright (r, p, m_1(s_1[r_1])) \cdot (q, p, m_2(s_2[r_2])) \cdot \epsilon \rightarrow P \{s_2[r_2]/x\} \mid (r, p, m_1(s_1[r_1])) \cdot \epsilon \]
The congruence relation $\equiv$ moves the message $m_2$ to the head of the queue, allowing to fire $[\text{R-Abs}]$. Hence, the session queue behaves as a set of unidirectional FIFO buffers, delivering messages between each pair of roles, akin to the TCP protocol.

## D ASYNCHRONOUS MULTIPARTY SESSION TYPES

We now extend the type system of §2 to the asynchronous calculus of §C. We reuse and overload definitions and notation from §2.2. We will prove subject reduction and session fidelity results with our new async type system (§F).

To type queues in the asynchronous calculus, we need **queue types** (Def. D.1).

**Definition D.1.** The **queue and session/queue types** are: (with $S$ from Def. 2.4)

$$M := p!m(S) \cdot M \mid \epsilon$$

(Session/queue types) $\tau := S \mid M \mid (M; S)$

The **congruence relation $\equiv$ for session/queue types $\tau$** is inductively defined as:

$$\begin{align*}
\text{Subtyping for session/queue types} & \text{ extends Def. 2.5 as:} \\
S & \equiv S' \iff \text{if } p \neq q \text{ then} \\
M & \equiv M' \iff S \equiv S' \\
M & \leq M' \iff (M; S) \leq (M'; S')
\end{align*}$$

Queue types are sequences of **message types** $p!m(S)$ having recipient $p$, label $m$, and payload type $S$ (omitted when $S = \text{end}$). A **session/queue type** can be a session type, a queue type, or a session/queue types pair: the latter describes queued messages (sent “in the past,” not yet received) and channel usage (that a process will fulfil “in the future”). The congruence $\equiv$ reorders queued messages with different recipients, like $\equiv$ in Fig. 7.

**Definition D.2.** An **async MPST typing context** is a partial mapping defined as:

$$\Gamma := \Gamma, s[p] : \tau \mid \Gamma, x : S \mid \emptyset$$

The **composition** $\Gamma_1, \Gamma_2$ is defined iff $\forall c \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) : \Gamma_1(c) = M$ and $\Gamma_2(c) = S$ and for all such $c$, we postulate $(\Gamma_1, \Gamma_2)(c) = (M; S)$. We extend $\equiv$ (from Def. D.1) to typing contexts as: $\Gamma \equiv \Gamma'$ iff $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and $\forall c \in \text{dom}(\Gamma) : \Gamma(c) = \Gamma'(c)$.

The relation $\Gamma \leq \Gamma'$ follows Def. 2.6, but using session/queue subtyping from Def. D.1.

Unlike Def. 2.6, in Def. D.2 above we have that: (1) channels with role map to session/queue types (but variables $x$ still map to session types, only); and (2) the **context composition** $\Gamma_1, \Gamma_2$ allows the domains of $\Gamma_1$ and $\Gamma_2$ to overlap on some $c$ — but only if $c$ maps to a session type in one context, and a queue type in the other. This can only occur if $c = s[p]$ (for some $s, p$), i.e., $c$ is not a variable $x$; then, $(\Gamma_1, \Gamma_2)(c)$ yields the combined session/queue type. The async typing rules in Fig. 8 induce the judgement:

$$\Theta \cdot \Gamma \vdash_S P \quad \text{where } S \text{ is a set of sessions } \{s_1, \ldots, s_n\}, \text{omitted when empty} \quad (14)$$
\[
\Theta \cdot \Gamma \vdash P \quad \text{(from Fig. 2, replacing rules \([\text{TA-\|}]/[\text{TA-v}]\) with \([\text{TA-\|}]/[\text{TA-v}]\))}
\]

\[
\Theta \cdot \Gamma_1 \vdash_{\mathcal{S}_1} P_1 \quad \Theta \cdot \Gamma_2 \vdash_{\mathcal{S}_2} P_2 \quad \mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset
\]

\[
\Gamma' = \{s[p]; s_p\}_{p \in I} \quad \text{s \notin \Gamma} \quad \varphi(\Gamma')
\]

\[
\Theta \cdot \Gamma \vdash_{\mathcal{S}, s} (\nu s; \Gamma') P
\]

\[
\Theta \cdot \Gamma \vdash s \triangleright \sigma \quad \Gamma' \vdash s' [r]: s
\]

where \(\varphi\) is a typing context property

Fig. 8. Asynchronous MPST typing rules: processes (top) and queues (bottom).

Unlike the sync MPST judgement (5), the context \(\Gamma\) of (14) is asynchronous (Def. D.2); further, (14) includes a set of sessions \(\mathcal{S}\) to track \(P\)'s queues; e.g., the parallel rule \([\text{TA-\|}]\) types parallel processes by combining their contexts, and requiring their session queues not to overlap \((\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset)\): this rejects processes with multiple queues per session, like \(Q \mid s \triangleright \sigma \mid s \triangleright \sigma'\). We will use \(\mathcal{S}\) in a more sophisticated way later, in §F. The lifting rule \([\text{TA-Lift}]\) types queueless processes, by lifting the synchronous typing judgement in Fig. 2. The session restriction rule \([\text{TA-v}]\) is akin to \([\text{TA-v}]\) (Fig. 2), but also removes the restricted \(s\) from the set of sessions; the typing context property \(\varphi\) is defined depending on the underlying proof strategy for subject reduction, with considerations similar to those highlighted in §2.3: we discuss the classic async MPST approach (and its issues) in §E, and our novel approach in §F.

The remaining rules are for typing queues. We have two queue rules: \([\text{TA-e}]\) types an empty queue \(s \triangleright \epsilon\) with the empty context; \([\text{TA-\sigma}]\) types a non-empty queue by inserting a message type in \(\Gamma\) using \(\leftrightarrow\), that might \((a)\) prepend the message to a queue type in \(\Gamma\), or \((b)\) add a queue-typed entry to \(\Gamma\), if not present.

**Example D.3.** The queue typing rules produce judgements like the following:

\[
\Theta \cdot \Gamma, s[r]: s\{p\} | \sum m_2(x). P \mid s \triangleright (r, p, m_1(s_1[r_1])) \cdot (q, p, m_2(s_2[r_2])) \cdot \epsilon
\]

\[
\Theta \cdot \Gamma, s[q]: s\{p\} | \sum m_2(x). P \mid s \triangleright (q, p, m_2(s_2[r_2])) \cdot \epsilon
\]

Note that \(s[p]\) has a session type (matching the process), while queued messages are typed by assigning them to their sender role, thus giving queue types to \(s[r]\) and \(s[q]\).

Async contexts reduce by Def. D.4 below: the definition is standard, except for the addition of transition labels. Unlike Def. 2.8, types interact in two phases: first, messages are queued (\([\text{\Gamma-AMsg}]\)); then, they are consumed (\([\text{\Gamma-AComm}]\)).

**Definition D.4.** Let \(\alpha\) have the form \(s\{p\} m\) or \(s\{p\} q\). The async typing context transition \(\alpha\) is inductively defined by the following rules, up-to congruence \(\equiv\) (Def. D.1), plus rules \([\text{\Gamma-\|}]\) and \([\text{\Gamma-Cong}]\) (Def. 2.8):

\[
\Theta \cdot \Gamma, s[p]: s\{m\} \quad \text{\([\text{\Gamma-AMsg}]\)}
\]

\[
\Theta \cdot \Gamma, s[q]: s\{m\} \quad \text{\([\text{\Gamma-AComm}]\)}
\]
The asynchronous reduction $\Gamma \rightarrow \Gamma'$ is defined iff $\Gamma \xrightarrow{\alpha} \Gamma'$ for some $\alpha$.

In Def. D.4, both transition rules and labels can be seen as different forms of synchronisation:
- $s.p|q;m$ denotes an interaction between a session type and its queue, with the addition of a pending message $m$ sent from $p$ to $q$, on session $s$;
- $s.p.q;m$ (that reuses notation from Def. 2.8) denotes the interaction between a queue type and a recipient session type, with the reception by $q$ of a queued message $m$ previously sent by $p$ on session $s$.

When the transition label is immaterial, we use the reduction $\rightarrow$.

E PROBLEMS OF CLASSIC ASYNCHRONOUS MPST

We now outline the issues of classic asynchronous MPST, to overcome them in §F. We summarise the technical definitions in Table 4.

Similarly to §3, classic async MPST require “asynchronously consistent” (or “a-consistent”) typing contexts. Therefore, rule \textbf{[TA-\nu]} in Fig. 8 is instantiated as follows:

\[
\Gamma' = \{s[p]_{p\in I}\} \quad s \notin \Gamma \quad \text{a-consistent}(\Gamma') \quad \Theta \cdot \Gamma, \Gamma' \vdash_S P
\]

\[
\Theta \cdot \Gamma \vdash_{S;\delta} (\nu s;\Gamma') P
\]

Async consistency (Def. E.2) caters for message queues by building upon (and further complicating) Fig. 3, thus inheriting its problems (cf. §3.1) and limitations (cf. §3.2). Moreover, the presence of queues and queue types, and their interaction with consistency, introduce two further difficulties, that complicate the classic subject reduction statement:

Empty Queues and “Missing” Reductions. The typing rules \textbf{[TA-\epsilon]/[TA-\sigma]} never map channels with role to empty queue types: this is visible in Ex.D.3 and Thm.L.5(6). E.g., take a typed process with empty queues $P = s[p][q] \oplus m(s'[q']) \cdot 0 | s \triangleright \epsilon$. By Fig. 8:

\[
\Theta \cdot \Gamma, s[p].q@m(S).\text{end} \vdash_0 s[p][q] @ m(s'[q']) \cdot 0 \quad \Theta \cdot \Theta \vdash_{\{s\}} s \triangleright \epsilon
\]

where $\Gamma = s[p]:q@m(S).\text{end}, s'[q']:]S$

Note that $\Gamma$ maps $s[p]$ to a session type without queue. Now, $P$ reduces as:

\[
(\text{Def. E.1 (Partial Asynchronous Projection). The message queue for } p \text{ in } M, \text{ written } M(p), \text{ is:})
\]
\[
(p!m(S) \cdot M)(p) = p!m(S) \cdot (M(p)) \quad (q!m(S) \cdot M)(p) = M(p) \quad (\text{if } p \neq q) \quad \epsilon(p) = \epsilon
\]

The queue prefixing of $M$ to $H$ is the partial type:

\[
p!m(S) \cdot M \bullet H = \oplus m(S). (M \bullet H) \quad \epsilon \bullet H = H
\]

The projection of $\tau$ onto $p$, written $\tau|_p$, is a partial session type defined as Def. 3.6 if $\tau = S$, and:

\[
M|_p = M(p) \cdot \text{end} \quad (M|_S)|_p = M(p) \cdot (S|_p)
\]

\[
\text{Definition E.2. } \Gamma \text{ is asynchronously consistent, written a-consistent}(\Gamma), \iff \forall s, p, q, \tau, \tau': \Gamma = \Gamma', s[p]:\tau, s[q]:\tau' \implies \tau|_q \subseteq \tau'|_p
\]

\text{Table 4. Classic async MPST consistency. These definitions build upon Fig. 3, and are not necessary in our new async MPST theory.}
\[ P \rightarrow P' = 0 \mid s \triangleright (p, q, m(s'[q'])) \cdot \epsilon \]

with \( \Theta \cdot \Gamma' \vdash_{(s)} P' \) and \( \Gamma' = s[p] \cdot (q!m(S) \cdot \epsilon; \text{end}), s'[q'] : S \)

Here, \( \Gamma' \) maps \( s[p] \) to a session type paired with a (non-empty) queue. Hence, \( \Gamma \) in (15) cannot match the process transition by reducing to \( \Gamma' \). By Def. D.4, the “missing” type reduction would be allowed if \( \Gamma \) mapped \( s[p] \) to the pair \((\epsilon; q@m(S), \text{end})\).

Async Consistency vs. Context Splits. Async consistency has a limitation w.r.t. synchronous consistency: it does not satisfy desideratum (D2) in §2.3. In fact, when the parallel typing rule [TA-l] (Fig. 8) splits a typing context, we might have:

\[ \text{a-consistent}(\Gamma_1, \Gamma_2) \nRightarrow \text{a-consistent}(\Gamma_1) \quad (16) \]

because \( \Gamma_1 \) might lose queue types in a way that breaks consistency. E.g., if we take:

\[ \Gamma_1 = \begin{cases} \{s[p] : q @ m_2, S', s[q] : p & m_1, p & m_2, S''\} & \Gamma_2 = s[p] : q! m_1 \cdot \epsilon \\
& (m_1 \neq m_2) \end{cases} \quad (17) \]

then \( \Gamma_1, \Gamma_2 \) is consistent, but \( \Gamma_1 \) is not, due to the mismatching output \( m_2 \) and input \( m_1 \). Consequently, async consistency is not preserved across typing derivations, and cannot be used in an induction hypothesis.

E.g., consider the process \( P \) below (slightly simplified by omitting irrelevant message payloads). Its sub-process \( P_p \) (who plays role \( p \)) has already sent a queued message \( n_1 \) and is about to send \( m_2 \) to \( q \), while the sub-process \( P_q \) (who plays role \( q \)) can receive both messages:

\[ P = P_p \mid P_q \mid s \triangleright (p, q, m_1) \cdot \epsilon \]

\[ P_p = s[p][q] \odot m_2, P' \]

\[ P_q = \text{def} \ X(x) = x[p] \sum m_1 \cdot x[p] \sum m_2 \cdot Q \text{ in } X(s[q]) \]

Note that we have \( P_q \rightarrow P''_q \), by expanding the call to \( X \) (rules [R-X] and [R-Crx] in Fig. 1); therefore, by [R-Crx], we also have \( P \rightarrow P' = P_p \mid P_q' \mid s \triangleright (p, q, m_1) \cdot \epsilon \). Now, if \( P \) is well-typed, we have:

\[ \Theta \cdot \Gamma_1 \vdash_{\emptyset} P_p \mid P_q \mid \Theta \cdot \Gamma_2 \vdash_{(s)} s \triangleright (p, q, m_1) \cdot \epsilon \]

where

\[ \Gamma_1 = \begin{cases} \{s[p] : q @ m_2, S', s[q] : p & m_1, p & m_2, S''\} \\
& \Gamma_2 = s[p] : q! m_1 \cdot \epsilon \end{cases} \]

Note that \( \Gamma_1, \Gamma_2 \) is consistent, but \( \Gamma_1 \) is not (due to the mismatching output of \( m_2 \) and input of \( m_1 \)). Thus, if we try to prove subject reduction for \( P \rightarrow P' \) by requiring an a-consistent typing context, we cannot apply the induction hypothesis on the premise \( P_p \mid P_q \rightarrow P_p \mid P_q' \).

Due to the issues above, the classic async subject reduction statement reads [Coppo et al. 2015a, Lemma 1]:

\[ \text{If } \Theta \cdot \Gamma \vdash_{S} P \text{ and } \exists \Gamma_0 \text{ such that } \Gamma, \Gamma_0 \text{ a-consistent and } P \rightarrow P', \]

then \( \exists \Gamma', \Gamma_0' \text{ a-consistent such that } \Gamma, \Gamma_0 \rightarrow \Gamma', \Gamma_0' \text{ and } \Theta \cdot \Gamma' \vdash_{S} P' \quad (18) \]

Intuitively, the statement solves the above issues by adding the typing context \( \Gamma_0 \), containing queue types that restore “missing” reductions and consistency.

In §F, we avoid these complications, and obtain more general results, by developing a new async MPST theory, that extends our new theory in §4 with novel semantics for asynchronous typing context, and a novel subject reduction statement and proof technique.
F  GENERAL ASYNCHRONOUS MULTIPARTY SESSION TYPE SYSTEM

We now present our async MPST type system. As in §4, it is modular, parametric w.r.t. an async safety property $\varphi$ (Def. F.2); to eschew the classic MPST difficulties summarised in §E, in the following we contribute a novel proof technique for asynchronous subject reduction, based on a novel handling of queue types:

1. we define a smarter typing context reduction $\rightarrow_S$ (Def. F.1) using the session set $S$;
2. we define async typing context $S$-safety (Def. F.2) by exploiting $\rightarrow_S$ above;
3. we develop a new subject reduction statement (Thm. F.6), simpler than (18), by exploiting the fact that our $S$-safety handles queueless type reductions, and survives context splits (Lemma F.5).

Definition F.1. The async typing context $S$-transition $\alpha \rightarrow_S$ is inductively defined by the rules below, up-to $\equiv$ (Def. D.2):

- $[\text{F-BASE}]$ $\Gamma \xrightarrow{\alpha} \Gamma'$ implies $\Gamma \xrightarrow{\alpha} S \Gamma'$
- $[\text{F-DMsg}]$ $s[p]:q \&_{I \in I} m_i(S_i).S'_i \xrightarrow{spq_\equiv k} s[p]:q!(m_k(S_k)\cdot \varepsilon; S'_k)$ if $s \in S$, $k \in I$
- $[\text{F-DCom}]$ $s[p]:q!m_k(S_k)\cdot \varepsilon, s[q].p \&_{I \in I} m_i(T_i).T'_i \xrightarrow{spq_\equiv k} s[q].T'_k$ if $s \in S$, $k \in I$, $S_k \leq T_k$

plus rules $[\text{F-LVar} ]$ and $[\text{F-Concl} ]$ (Def. 2.8), replacing $\rightarrow$ with $\rightarrow_S$.

We write $\Gamma \xrightarrow{\alpha}_S$ iff there is $\Gamma'$ such that $\Gamma \xrightarrow{\alpha}_S \Gamma'$. We $\Gamma \rightarrow_S \Gamma'$ iff $\Gamma \xrightarrow{\alpha}_S \Gamma'$ for some $\alpha$.

In Def. F.1, rule $[\text{F-BASE}]$ says that any async reduction $\Gamma \xrightarrow{\alpha} \Gamma'$ (Def. D.4) is matched by $\rightarrow_S$. To support reductions of session types without queues, $[\text{F-DMsg}]$ allows an internal choice to reduce by creating a queue type carrying its output, and $[\text{F-DCom}]$ allows a queue type to disappear when its last message is consumed. Crucially, $[\text{F-DMsg}]$ and $[\text{F-DCom}]$ only apply for sessions in $S$; otherwise, $\rightarrow_S$ matches $\rightarrow$, i.e., queueless types do not reduce.

Definition F.2 (Asynchronous Safety). $\varphi$ is an $S$-safety property on typing contexts iff:

- $[\text{SA-Al}]$ $\varphi(\Gamma, s[p]:q \&_{I \in I} m_i(S_i).S'_i, s[q]:M)$ and $M \equiv p!m(T)\cdot M'$ implies $\exists k \in I: m_k = m, T \leq S_k$;
- $[\text{SA-L}]$ $\varphi(\Gamma, s[p]:\mu t.S)$ implies $\varphi(\Gamma, s[p]:S(\mu t.S/t))$;
- $[\text{SA-R}]$ $\varphi(\Gamma)$ and $\Gamma \rightarrow_S \Gamma'$ implies $\varphi(\Gamma')$.

We say $\Gamma$ is asynchronously $S$-safe, or a-safe$_S(\Gamma)$, iff $\varphi(\Gamma)$ for some $S$-safety property $\varphi$.

The notion of $S$-safety in Def. F.2 is akin to Def. 4.1, but caters for asynchrony: if a queue has a top-level message from $p$ to $q$, and $q$ is trying to receive from $p$, then their messages must be compatible and allow them to reduce, by Def. D.4. Note that clause $[\text{SA-R}]$ uses $\rightarrow_S$; i.e., if a queueless type belongs to a session in $S$ it can reduce, otherwise is stuck and ignored, as shown in Ex.F.3.

Example F.3. Take $\Gamma_1$ and $\Gamma_2$ from (16) above. We have a-safe$_{\{s\}}(\Gamma_1, \Gamma_2)$, but a-safe$_{\{s\}}(\Gamma_1)$ does not hold: $\Gamma_1$ reduces by $\rightarrow_{\{s\}}$, queuing message $m_2$, and violating clause $[\text{SA-\&L}]$ (Def. F.2). Instead, a-safe$_{\varphi}(\Gamma_1)$ holds, since $\Gamma_1 \rightarrow P_{\varphi}$, and $\Gamma_1$ (vacuously) satisfies Def. F.2.

We now have all the ingredients for our general async type system.

Definition F.4. The general asynchronous MPST typing judgement is induced by the rules in Fig. 8 — with rule $[\text{TA-v}]$ restricted as follows:

\[
\Gamma' = \{s[p]:S_p\}_{p \in I} \quad \varphi(\Gamma') \quad s \notin \Gamma \quad \Theta \cdot \Gamma, \Gamma' \rightarrow_S P \quad \text{[TAGEN-v]} \quad \text{where } \varphi \text{ is an } \{s\}-\text{safety property}
\]

We write "$\Theta \cdot \Gamma \vdash P$ with $\varphi$" to specify how to instantiate $\varphi$ in rule $[\text{TAGEN-v}]$ above. When "with $\varphi$" is omitted, then the instantiation is $\varphi = \text{a-safe}_{S_U}$ (i.e., the largest $S_U$-safety property, cf. Def. F.2) where $S_U$ is the set of all sessions.
As in Def. 4.6, Def. F.4 provides a novel foundation for asynchronous MPST, but has just one visible change w.r.t. the classic multiparty session typing rules: the rule for session restriction, that uses a parametric, behavioural (rather than syntactic) property on \( \Gamma' \). Note that, crucially, we exploit the judgement’s session set to determine \( S \) for async \( S \)-safety (Def. F.2); by using the same \( S \) for \( \rightarrow_S \) (Def. F.1), we are able to formalise and prove asynchronous subject reduction, type safety, and session fidelity, as follows. Crucially, Thm. F.6 (that provides the precise formal statement of Thm. 7.1) uses Lemma F.5 as a weak form of desideratum \( \text{(D2)} \) in \( \S2.3 \): it provides fine-grained splits of async typing contexts (again, depending on the session set \( S \)) that preserve safety along the subject reduction proof.

**Lemma F.5.** Let \( \text{a-safe}_S(\Gamma) \): then, \( \text{a-safe}_{S\downharpoonright\sigma}(\Gamma) \); and if \( \Gamma = \Gamma', s[p] : S \), then \( \text{a-safe}_S(\Gamma') \).

**Theorem F.6 (Asynchronous Subject Reduction).** Assume \( \Theta \cdot \Gamma \vdash_S P \) with \( \Gamma \) \( S \)-safe. Then, \( P \rightarrow P' \) implies \( \exists \Gamma' \) \( S \)-safe such that \( \Gamma \rightarrow_S \Gamma' \) and \( \Theta \cdot \Gamma' \vdash_S P' \).

**Corollary F.7 (Async Type Safety).** If \( \Theta \cdot \emptyset \vdash P \) and \( P \rightarrow^* P' \), then \( P' \) has no errors.

**Theorem F.8 (Async Session Fidelity).** Let \( \Theta \cdot \Gamma \vdash_S P \), with \( P \equiv \left( \prod_{p \in I} P_p \right) \mid s \triangleright \sigma \), and each \( P_p \) either being \( \emptyset \) (up-to \( = \)), or only playing role \( p \) in \( s \). Then, \( \Gamma \rightarrow_S \Gamma' \), \( P \rightarrow^* P' \) and \( \Theta \cdot \Gamma' \vdash_S P' \), with \( P' \equiv \left( \prod_{p \in I} P'_p \right) \mid s \triangleright \sigma' \) and each \( P'_p \) either being \( \emptyset \) (up-to \( = \)), or only playing role \( p \) in \( s \).

Finally, similarly to the synchronous case (Thm. 4.11), also asynchronous type checking is decidable, when instantiated with a decidable safety property.

**Theorem 7.2.** If \( \varphi \) is decidable, then \( \"\Theta \cdot \Gamma \vdash_S P \) with \( \varphi \) \( \" \) is decidable.

**G FROM ASYNC TYPING CONTEXT PROPERTIES TO PROCESS PROPERTIES**

As in \( \S5 \), we now present several properties reinforcing \( \text{a-safe}_S \), compare them, and use them to instantiate \( \varphi \) in our type system, to predict and constrain the run-time behaviour of processes. However, under asynchrony, we need to address an additional challenge: since queue types are unbounded, an asynchronous typing context \( \Gamma \) can induce an infinite state transition system, making its properties undecidable (unlike Thm. 5.13).

Def. G.2 below is the async version of the typing context properties discussed in \( \S5.3 \). The key difference is that Def. G.2 checks queued message types, instead of internal choices. But first, we need to formalise the asynchronous version of Def. 5.5 (fair traversals), in Def. G.1 below.

It’s purpose is similar: find a set of roles that can interact and always reach a target state, under “fair scheduling”; but in Def. G.1, we also choose the “right time” to fire queuing and reception transitions: this allows to ignore unfair executions where a recursive output is fired infinitely, and enqueues infinitely many messages, without giving a chance to the recipient to consume them.

**Definition G.1 (Asynchronous fair traversal set).** Let \( X, Y \) be sets of asynchronous typing contexts. We say that \( X \) is a fair traversal set for sessions \( S \) with targets \( Y \) iff \( X \) is closed under the rules:

\[
\begin{align*}
\Gamma \in Y & \quad \Rightarrow \quad \Gamma \in X \quad \text{[TSA-TARGET]} \\
\exists S \in S, p, q : \exists m : \left( \Gamma \xrightarrow{s.p.q^m}_S \Gamma' \right. \text{ and } \left. \Gamma \xrightarrow{s.p.q^m}_S \Gamma' \text{ implies } \Gamma' \in X \right) \\
\end{align*}
\]
We can now formalise Def. G.2. Most definitions match those in Fig. 5; the additional item (7) places a bound on the length of type queues: it will be useful later, for Thm. G.5.

**Definition G.2 (Properties of Asynchronous Typing Contexts).** We write a-$\mathsf{end}_S(\Gamma)$ iff $\forall s[p] \in \mathsf{dom}(\Gamma)$ with $s \in S$, $\Gamma(s[p]) \in \{S, \epsilon, (e; S) | S \leq \mathsf{end}\}$.

1. We say that $\Gamma$ is $S$-$\mathsf{deadlock-free}$, or a-$\mathsf{df}_S(\Gamma)$, iff $\Gamma \rightarrow_S \Gamma' \rightarrow_S$ implies $\mathsf{end}_S(\Gamma')$.

2. We say that $\Gamma$ is $S$-$\mathsf{terminating}$, written a-$\mathsf{term}_S(\Gamma)$, iff $\Gamma$ is $S$-deadlock-free, and there is $k \in \mathbb{N}$ such that for all $n \geq k$, $\Gamma = \Gamma_0 \rightarrow_S \Gamma_1 \rightarrow_S \cdots \rightarrow_S \Gamma_n$ implies $\mathsf{end}_S(\Gamma_n)$.

3. We say that $\Gamma$ is $S$-$\mathsf{never-terminating}$, written a-$\mathsf{nterm}_S(\Gamma)$, iff $\Gamma \not\rightarrow_S \Gamma'$ implies $\Gamma' \not\rightarrow_S$.

4. $\phi$ is an $S$-$\mathsf{live}$ property on asynchronous typing contexts iff:

   \begin{align*}
   \text{[LA-\&] } & \phi(\Gamma, s[p]; S) \text{ with } S = q_i \epsilon m_i(S_i).S_i' \text{ implies } \exists i \in I : \exists \Gamma' : \exists s[p].S \rightarrow_S \Gamma', s[p].S_i' \\
   \text{[LA-\&\&] } & \phi(\Gamma, s[p]; M) \text{ with } M \equiv q_i m(S).M' \text{ implies } \exists \Gamma' : \exists s[p].M \rightarrow_S \Gamma', s[p].M'
   \end{align*}

   plus clauses [SA-$\mu$] and [SA-$\rightarrow$] from Def. F.2. We say that $\Gamma$ is asynchronously $S$-live, written a-$\mathsf{live}_S(\Gamma)$, iff $\phi(\Gamma)$ for some $S$-live property $\phi$.

5. $\phi$ is an $S$-$\mathsf{live}^*$ property iff:

   \begin{align*}
   \text{[LA-\&\&\&] } & \text{clause [LA-\&] above; moreover, } \Gamma \text{ belongs to some async fair traversal set } \mathcal{X} \text{ for sessions } S \\
   \text{[LA-\&\&\&\&] } & \text{with targets } Y \text{ (Def. G.1)} \text{ such that, } \forall \Gamma_i \in Y, \text{ we have } \Gamma_i = \Gamma''', s[p].S_i' \text{ (for some } \Gamma''', i \in I)
   \end{align*}

   plus clauses [SA-$\mu$] and [SA-$\rightarrow$] from Def. F.2. We say $\Gamma$ is asynchronously $S$-live*, or a-$\mathsf{live}^*_S(\Gamma)$, iff $\phi(\Gamma)$ for some $S$-live* property $\phi$.

6. $\phi$ is an $S$-$\mathsf{live}^{*+}$ property iff:

   \begin{align*}
   \text{[LA-\&\&\&\&\&] } & \text{clause [LA-\&\&] above; moreover, there is } n \in \mathbb{N} \text{ such that if } \Gamma = \Gamma_0 \rightarrow_S \Gamma_1 \rightarrow_S \cdots \rightarrow_S \Gamma_n, \\
   \text{[LA-\&\&\&\&\&\&] } & \exists i < n \text{ such that } \Gamma_i \rightarrow_S \Gamma_{i+1} = \Gamma''', s[p].S_i' \text{ (for some } \Gamma''', i \in I)
   \end{align*}

   plus clauses [SA-$\mu$] and [SA-$\rightarrow$] from Def. F.2. We say $\Gamma$ is asynchronously $S$-live**, or a-$\mathsf{live}^{*+}_S(\Gamma)$, iff $\phi(\Gamma)$ for some $S$-live** property $\phi$.

7. $\Gamma$ is $k$-bounded (w.r.t. $S$), or a-$\mathsf{bound}_{S,k}(\Gamma)$, iff $k \in \mathbb{N}$ and $\Gamma \rightarrow_S \Gamma'$, $s[p].M$ implies $|M| \leq k$. $\Gamma$ is bounded (w.r.t. $S$), or a-$\mathsf{bound}_S(\Gamma)$, iff $\exists k$ finite: a-$\mathsf{bound}_{S,k}(\Gamma)$.

Lemma G.3 below is the async version of Lemma 5.9. Items 8 and 9 show that boundedness does not imply any other property, and is only implied by termination.

**Lemma G.3.** For all $\Gamma$, letting $S = \{s | \exists p : s[p] \in \mathsf{dom}(\Gamma)\}$, we have:

1. a-$\mathsf{consistent}(\Gamma) ⇔ a-\mathsf{safe}_S(\Gamma)$;
2. a-$\mathsf{live}_S(\Gamma) ⇔ a-\mathsf{safe}_S(\Gamma)$;
3. a-$\mathsf{term}_S(\Gamma) ⇔ a-\mathsf{df}_S(\Gamma)$;
4. a-$\mathsf{nterm}_S(\Gamma) ⇔ a-\mathsf{df}_S(\Gamma)$;
5. a-$\mathsf{consistent}(\Gamma) ⇔ ⇔ a-\mathsf{df}_S(\Gamma)$;
6. a-$\mathsf{term}_S(\Gamma) \land a-\mathsf{df}_S(\Gamma) ⇔ a-\mathsf{live}_S(\Gamma)$;
7. a-$\mathsf{term}_S(\Gamma) ⇔ a-\mathsf{live}_S^{+}(\Gamma)$;
8. a-$\mathsf{term}_S(\Gamma) ⇔ a-\mathsf{bound}_S(\Gamma)$;
9. a-$\mathsf{bound}_S(\Gamma) ⇔ a-\mathsf{safe}_S(\Gamma) \lor a-\mathsf{df}_S(\Gamma)$;
10. a-$\mathsf{live}^{+}_S(\Gamma) ⇔ a-\mathsf{live}^{+}_S(\Gamma) ⇔ a-\mathsf{live}_S(\Gamma)$. 
Example G.4. Ex.5.10 and Ex.5.11 also hold under the async properties of Def. G.2. Moreover:

(1) The typing context $\Gamma$ from Ex.2.7 (i.e., our example in §1) is bounded;

(2) $\Gamma_2$ from Ex.5.11 is not bounded: $s[p]$ can queue infinite outputs, before $s[q]$ consumes them;

(3) the “arbitrary typing context” $s[p]::q@foo(end), s[q]::p&bar(end)$, $s[r]::end$ from §2.3 is async bounded, but not async safe nor deadlock-free;

(4) $s[p]::\mu t.q@m_1.q&\mu m_2.t$, $s[q]::\mu t.p@m_2.p&\mu m_1.t$ is not a-consistent, and is deadlocked under synchronous reductions (Def. 2.8); however, under $\rightarrow_{(s)}$ (Def. D.4) it is async live+ (hence async safe) and bounded ($k = 2$). In fact, both $s[p]$ and $s[q]$ can reduce by enqueuing their initial output messages ($m_1$ and $m_2$), and then receiving the message enqueued by the other party;

(5) $s[p]::\mu t.q@m_1.q&\mu m_1.q&\mu m_2.t$, $s[q]::\mu t.p@m_2.p&\mu m_1.t$ (akin to (4) above) is live+, but not a-consistent nor bounded: in each loop, two messages $m_i$ are queued and only one is consumed.

Note that the diagram of Lemma G.3 includes $G$ and $G-$ from Lemma 5.9, i.e., the sets of contexts projected from a global type: they do not occur in the statement, but will be justified by Thm.G.6 and Remark G.7.

Decidability. Thm.G.5 below provides minimal decidability criteria for async properties.

**Theorem G.5.** $\forall \Gamma$, a-consistent($\Gamma$) is decidable. Furthermore, $\forall S, k$, a-bound$_{S,k}^+\Gamma$ is decidable; and if a-bound$_{S,k}^\Gamma$ holds, then $\Gamma$ has a finite state transition system, hence $a$-safe$_S\Gamma$, a-df$_S\Gamma$, a-term$_S\Gamma$, a-nterm$_S\Gamma$, a-live$_S\Gamma$, a-live$_{S,+}^+\Gamma$ and a-live$_{S,+}^\Gamma$ are decidable.

By Thm.G.5 and Thm.7.2, we obtain various decidable instances of our general asynchronous MPST type system. Such instances require either asynchronous consistency (as in classic async MPST), or the enforcement of a limit on the size of queue types, to ensure that the properties in Def. G.2 are decidable.

Besides such results, decidability of $\varphi$ (and type checking) is hampered due to the correspondence between session/queue types and Communicating Finite State Machine (CFSM) [Denielou and Yoshida 2013]: (1) a session/queue type $(M+S)$ is a CFSM with finite control $S$ and output queues $M$, and (2) an async $\Gamma$ is a system of interacting CFSMs. Unfortunately, safety and other properties in Def. G.2 (including the existence of a queue bound $k$) are undecidable for CFSMs [Brand and Zafirropulo 1983] [Genest et al. 2007], and $\Gamma$ can encode a Turing machine [Bartoletti et al. 2016, Thm. 2.5]. To address this problem, we cannot straightforwardly lift decidable synchronous properties to asynchrony, e.g.:

$\Gamma = s[p]::r@m_1.q@m_2$, $s[q]::p&m_1$ with $m_2 \neq m_3$

$\Gamma \rightarrow_{(s)} \rightarrow_{(s)} \Gamma' = s[p]::q!(m_2::r!(m_1::\epsilon)::end)$, $s[q]::p&m_3$

Note that $\Gamma$ is synchronously safe (Def. 4.1), but not asynchronously $\{s\}$-safe (Def. F.2): the outputs of $s[p]$ are queued, and $\Gamma'$ violates Def. F.2 ($\text{[sa-!]}$). Luckily, Thm.G.6 says that the session types/CFSMs correspondence allows to lift liveliness to asynchrony; and building upon this basis, we can also lift liveness+ to asynchrony (notice that the latter is a completely new result).

**Theorem G.6.** Let $\Gamma$ be a synchronous typing context (Def. 2.6); moreover, let $S = \{s \mid \exists p : s[p] \in \text{dom}(\Gamma)\}$. Then, live($\Gamma$) is decidable, and implies a-live$_S\Gamma$. Moreover, live+($\Gamma$) is decidable, and implies a-live$_{S,+}^+\Gamma$.

**Proof.** [Denielou and Yoshida 2013, Def. 4.2] and [Bocchi et al. 2015, Def. 4] define Multiparty Compatibility (MC) as a behavioural property based on $\Gamma$’s synchronous reductions; its definition corresponds to our synchronous liveness (Fig. 5); further, MC is decidable (as our Thm.5.13). Finally, [Denielou and Yoshida 2013, Thm 4.1] and [Bocchi et al. 2015, Thm. 6] say that if $\Gamma$ is live/MC, then queued outputs are eventually consumed, and external choices are eventually triggered, as in
async liveness (Def. G.2). This leads to the result on liveness+. For the result on liveness+, we further leverage the live/MC correspondence above to rule out the existence of communication loops that would not allow to build the fair traversal sets required by clauses [LA-\&-]\([LA-\&^+]\) of Def. G.2(S). For further details, see §N.

**Remark G.7.** With Theorems G.5, G.6, and 7.2, we can instantiate \(\varphi\) in Def. F.4 as described in §7 (methods (M1)–(M3)), to obtain various decidable type-checking methods for async processes. In particular, by Lemma 5.9(9) and Thm. G.6, we can use global types to project async live* typing contexts (hence the sets \(G\) and \(G\) in the diagram of Lemma G.3). Thus, such decidable instances of our type system subsume classic async MPST (that only cover \(\varphi=\alpha\)-consistent), and type the async version of our example in §1 — i.e., Remark 5.12 also applies to async MPST. Also note that we support protocols that are “inherently” asynchronous, and cannot be projected from any global type — cf. Ex. G.4(4),G(5).

To conclude, we show how the properties defined above influence process behaviours (Thm. G.9). These results are new, and intuitively similar to Thm. 5.15 (for synchronous MPST), modulo the presence of message queues.

**Definition G.8.** \(P\) is asynchronously deadlock-free iff \(P \rightarrow^* P' \not\rightarrow\) implies \(P' \equiv 0\). \(P\) is asynchronously terminating iff it is async deadlock-free, and \(\exists k\) finite such that, \(\forall n \geq k\), \(P = P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_n\) implies \(P_n \equiv 0\). \(P\) is asynchronously never-terminating iff \(P \rightarrow^* P'\) implies \(P'\rightarrow\). We say that \(P\) is asynchronously live* iff \(P \rightarrow^* P' \equiv \mathbb{C}[Q]\) implies:

1. if \(Q = s\upDownarrow \sigma\) with \(\sigma \in (p.q,m(s'\lceil r\rceil)')\cdot\sigma'\), then \(P' \rightarrow^* \mathbb{C}'[s\upDownarrow \sigma']\) for some \(\mathbb{C}'\);
2. if \(Q = c[q]\sum_{i \in I} m_i(x_i).Q_i'\), then \(P' \rightarrow^* \mathbb{C}'[Q_k'\{s'\lceil r\rceil/s_k\}]\) for some \(\mathbb{C}'\), \(k \in I, s', r\);

\(P\) is strongly asynchronously live iff \(P \rightarrow^+ P' \equiv \mathbb{C}[Q]\) implies:

1. item 1 above, and moveover, there is \(n\) finite such that, whenever \(P' = P_0' \rightarrow P_1' \rightarrow \cdots \rightarrow P_n'\), then for some \(j \leq n\) we have \(P_j' \rightarrow \mathbb{C}'[s\upDownarrow \sigma']\) (for some \(\mathbb{C}'\));
2. item 2 above, and moveover, there is \(n\) finite such that, whenever \(P' = P_0' \rightarrow P_1' \rightarrow \cdots \rightarrow P_n'\), then for some \(j \leq n\) we have \(P_j' \rightarrow \mathbb{C}'[Q_k'\{s'\lceil r\rceil/s_k\}]\) (for some \(\mathbb{C}'\), \(k \in I, s', r\)).

**Theorem G.9.** Assume \(0\cdot \Gamma \vdash_\mathcal{S} P \equiv \left(\bigcap_{p \in P} P_p\right) | s \upDownarrow \sigma\), with all \(P_p\) having guarded definitions and being 0 (up-to \(\equiv\)), or only playing \(p\) in \(s\). Then: (1) a-df\(\mathcal{S}(\Gamma)\) implies \(P\) is async deadlock-free; (2) a-term\(\mathcal{S}(\Gamma)\) implies \(P\) is async terminating; (3) a-term\(\mathcal{S}(\Gamma)\) implies \(P\) is async never-terminating; (4) a-live\(\mathcal{S}(\Gamma)\) implies \(P\) is async live. (5) a-live\(\mathcal{S}^+(\Gamma)\) implies \(P\) is strongly async live.

**Remark G.10.** The a-df\(\mathcal{S}(\Gamma)\), a-live\(\mathcal{S}(\Gamma)\) a-live\(\mathcal{S}^+(\Gamma)\) hypothesis used in Thm. G.9(1) are generally undecidable, but they are implied by (decidable) synchronous liveness/liveness+ (by Thm. G.6 and Lemma G.3) — which in turn, can be verified using the model checking techniques discussed in §6.

Moreover, the premises of items (1)–(5) of Thm. G.9 can be decided by first checking whether \(\Gamma\) has \(k\)-bounded queues, given some \(k\) (by Thm. G.5). Furthermore, as highlighted in Remark G.7, global types are a (decidable) way to project typing contexts that are a-live*, and thus, ensure that typed processes are live (by Thm. G.9(4)). These results are more general than those achievable under classic MPST, because they do not require the existence of global types, nor consistency of \(\Gamma\).

**H ADDITIONAL RELATED AND FUTURE WORK**

The main related works are discussed in §8. In this section, we discuss other approaches for formalising protocols and types processes, and show why they cannot handle our examples; we also discuss other papers and extensions that are specifically related to our general asynchronous MPST theory.
H.1 Conversation Types

Conversation types where proposed by [Caires and Vieira 2009, 2010], as a typing discipline for ensuring that processes implement given protocols. They share various goals and fundamental ideas with session types (including a notion of *duality*), but their technical development is rather different; among their main features, the flexible assignment of protocol roles to processes, and the capability of letting processes dynamically join existing sessions; both features allow to model forms of multiparty interaction.

Conversation types do not have consistency requirements that can be directly compared with those of classic MPST; still, they have several type-level operators and constraints that remind syntactic duality, and do not support our examples in Fig. 4.

Consider, e.g., our opening example (§1). The conversation type corresponding to the global type $G$ in (1) is (19) below, where $\tau m(T)$ denotes a communication where two parties exchange message $m$ carrying a value of type $T$, and $\oplus$ denotes an internal choice:

$$
\oplus \{ rlogin.rpasswd(Str).rauth(Boolean),
\}
\{ rcancel.rquit \}
$$

(19)

Note the lack of explicit roles in (19). This is a distinguishing feature of conversation types: they can be decomposed in various ways, using a *merge relation* $\bowtie$ [Caires and Vieira 2010, Def. 3.11]. In order to match the scenario in §1, we would need to decompose (19) into three types, combined by $\bowtie$: this is the only way to type three separate processes (for the service, client, and authorisation server), using rule [Par] in [Caires and Vieira 2010, Fig. 9]. Therefore, we would like to decompose (19) as follows, where $!/?$ denote output/input messages, and $\&$ is an external choice:

$$
\bowtie \{ !login.?auth(Boolean),
\}
\& \{ !cancel,?
\cancel .?quit \}
\bowtie \{ ?passwd(Str).!auth(Boolean),
\}
\& \{ ?quit \}
$$

(20)

However, the merging in (20) is undefined: by [Caires and Vieira 2010, Def. 3.11], the rightmost external choice (i.e., the type of the authorisation server) can only be merged with an internal choice $\oplus$ having dual output messages (rules [PLAIN-l]/[PLAIN-r] in [Caires and Vieira 2010, Fig. 8]) — but the other types in (20) do not satisfy this requirement: this is because the desired output messages are prefixed by (i.e., depend on) choices and other interactions.\(^4\) Alternatively, one might try to adjust (19), and leverage *directions* and their *projection* [Caires and Vieira 2010, Def. 3.7], aiming at an implementation that starts a binary conversation between the service and client, and involves the authorisation server at a later stage. This would be a significant change; besides, it would still require to decompose the overall protocol using $\bowtie$, with the limitations described above.

As a future research direction, it would be interesting to investigate whether our approach based on safety (Def. 4.1) can provide a new foundation for conversation types, replacing its merge relation with a more flexible alternative.

H.2 Global Types Semantics and Choreographic Programming

Various works investigate the semantic properties of global types (also called *choreographies*) and their projections: for example, [Castagna et al. 2012; Lanese et al. 2008]. Unlike the present work, such papers do not investigate type systems (i.e., do not use projections as types), nor type safety results; hence, they do not address the issues described in §2.3, nor develop results like ours.

\(^4\)This reminds the reason why, in classic MPST, some partial projections of the same protocol are undefined, cf. §3.1. This suggests a common fundamental limitation: the authorisation protocol in §1, and all other examples in Fig. 4, are inherently multiparty, and cannot be cleanly decomposed into sets of binary interactions; however, a binary decomposition is required both in classic MPST (via partial projection/consistency), and in conversation types (via merging). Our new MPST theory (§4) does not have this requirement.
On a related line of research, [Carbone and Montesi 2013] propose choreographic programming: an approach where concurrent and distributed application are directly developed as choreographies, using a language that reminds a global type specification (Def. 3.2) enriched with programming constructs (e.g., variables, data, conditionals). Choreographic programs are compiled by projecting them into endpoint processes (in a variant of the multiparty session π-calculus); once deployed and executed, such processes interact as specified in the original choreographic program, which is deadlock-free by construction.

The choreographic programming approach is somewhat “opposite” w.r.t. MPST, and does not address scenarios like our example in § 1. In fact, in [Carbone and Montesi 2013], distributed applications are developed as single programs, type-checked against a global type; unlike MPST, distributed components are not supposed to be developed independently, and separately verified against a desired interface (i.e., a session type). For example:

- with MPST, the authorisation server, the client and the service in § 1 can be developed and verified separately and independently, by different programmers, working on distinct codebases. Hence, the developer of the authorisation server only needs to type-check the implementation $P_a$ against the session type $S_a$ in (2) (cf. Ex.2.7); this is enough to guarantee correct interaction with other third-party processes, as long as their combined types are safe (cf. Def. 4.1);
- instead, in choreographic programming, the authorisation server, the client and the service in § 1 need to be written as a single combined “global program,” that would look like $G$ in (1). In many cases, this is not desirable or feasible.

A payoff of this restriction is that type system of [Carbone and Montesi 2013] does not encounter the difficulties of MPST systems: since processes are not type-checked separately against a desired interface, typing derivations like Ex.2.7 do not arise, and this avoids the issues illustrated in §2.3.

The limitations above are directly addressed in [Montesi and Yoshida 2013], who propose compositional choreographies: they develop a unified language for both choreographies and endpoint programs — and the latter are called partial choreographies, with the possibility of being separately developed, and reused. To this purpose, unlike [Carbone and Montesi 2013], [Montesi and Yoshida 2013] introduce parallel composition of (partial) choreographies, and communication via synchronisation (cf. (par) in Fig. 2, and rule $[C|\text{Sync}]$ in Fig. 3 of [Montesi and Yoshida 2013]). This leads to the introduction of session types, and typing contexts denoted with $\Delta$ [Montesi and Yoshida 2013, p. 432], and reductions similar to our Definitions 2.6 and 2.8, and a typing rule for parallel composition (rule $[\text{T|Par}]$ in [Montesi and Yoshida 2013, p. 432]) similar to our rule $[\text{T-|}]$ in Fig. 2.

The resulting typing derivations are similar to our Ex.2.7, and have issues like those discussed in §2.3, that require to constrain $\Delta$. The subject reduction statement [Montesi and Yoshida 2013, Thm. 1] does not show explicit constraints on $\Delta$ — but since the paper integrates classic MPST from Honda et al. [2008] and Deniélou et al. [2012], it should either (a) inherit the duality/consistency issues and limitations discussed in §3 (i.e., not supporting our opening example in §1, nor the other examples in Fig.4), or (b) implement a rather complex subject reduction proof strategy similar to the “non-classic” approach of Dezani-Ciancaglini et al. [2015] and Ghilezan et al. [2018] (discussed in §8.2). On the positive side, we believe that our new MPST theory can be used as a drop-in replacement for classic MPST in [Montesi and Yoshida 2013]: the resulting integration would support our examples, without significant changes to the rest of their paper.

### H.3 Asynchronous Subtyping

Our new asynchronous type system (even in its decidable instances) supports asynchronous protocols whose correctness depends on the capability of buffering messages, and consuming them
at a later time. This means that we can use typing contexts (and type-check processes) that interact correctly under asynchrony, but would deadlock under synchronous semantics: see Ex. G.4, cases (4) and (5). This feature is not supported by the classic async MPST theory, because its a-consistency requirement (Def. E.2) only accepts types (and processes) that interact dually under synchronous semantics, disregarding asynchronous message buffering.

To overcome this limitation of classic MPST, [Mostrous et al. 2009] introduced an asynchronous subtyping relation $\leq_a$ that allows to “anticipate” outputs w.r.t. inputs: for example, if $S = p \& m_1 \oplus p \& m_2$ and $S' = p \& m_2 \oplus p \& m_1$, we have $S \leq_a S'$; therefore, by using $\leq_a$ in Fig. 2 (rule $[\text{T-Sub}]$) and Fig. 8, a typed asynchronous process can use an $S$-typed channel according to $S'$, to first send $m_2$ (that is buffered at run-time) and then receive $m_1$, without causing deadlocks nor communication errors.  

Asynchronous subtyping has been further studied (for binary sessions) in [Chen et al. 2017, 2014], and later discovered to be undecidable in [Bravetti et al. 2017; Lange and Yoshida 2017]. We can seamlessly integrate $\leq_a$ in our new MPST theory by proving that if $a$-safe$_{\leq_a}$($\Gamma$) and $\Gamma \leq_a \Gamma'$, then $a$-safe$_{\leq_a}$($\Gamma'$); and to preserve Thm. G.9, we also need to prove that if $a$-live$^}_{\leq_a}$($\Gamma$) and $\Gamma \leq_a \Gamma'$, then $a$-live$^\prime_{\leq_a}$($\Gamma'$). However, the undecidability of $\leq_a$ would make type checking undecidable, thus falsifying Thm. 7.2; therefore, to preserve Thm. 7.2, we could only adopt decidable fragments of $\leq_a$ — and the known ones (studied in [Bravetti et al. 2017, 2018; Lange and Yoshida 2017]) are limited to binary sessions. Hence, integrating $\leq_a$ in our new theory would introduce a limited gain — especially because, as explained above, our new async MPST theory already supports asynchronous protocols. Therefore, we decided to leave the further study of multiparty asynchronous subtyping as future work.

### ADDITIONAL REFERENCES FOR THE APPENDIX


Luís Caires and Hugo Torres Vieira. 2009. Conversation Types. In ESOP. https://doi.org/10.1007/978-3-642-00590-9_21


---

5The asynchronous subtyping relation $\leq_a$ outlined here is inverted w.r.t. the one in [Mostrous et al. 2009], for the reasons explained in footnote 1.

Appendices — Part 2

Proofs
I SESSION INVERSION AND FIDELITY

Proposition I.1. If $P \equiv P'$, then $\text{fc}(P) = \text{fc}(P')$ and $\text{fv}(P) = \text{fv}(P')$.

Proof. By examining the cases where $P \equiv P'$ holds, and by applying the definition of $\text{fc}(-)$ and $\text{fv}(-)$.

Proposition I.2 (Normal Form). For all $P$, $P \equiv \text{def } \overrightarrow{D}$ in $(\forall s) P_1 | \ldots | P_n$, where $\forall i \in 1..n$, $P_i$ is either a branching, a selection, or a process call.

Proof. From [Coppo et al. 2015a, Proof of Thm. 1].

Proposition I.3. For all $S$, $S \leq \text{end}$ if and only if $\text{end} \leq S$.

Proof. Follows by Def. 2.5.

Lemma I.4 (Typing Inversion). Assume $\Theta \cdot \Gamma \vdash P$. Then:

1. $P = 0$ implies $\text{end}(\Gamma)$;
2. $P = \text{def } X(x_1:S_1, \ldots, x_n:S_n) = Q$ in $P'$ implies:
   (i) $\Theta \cdot X; S_1, \ldots, S_n \vdash \Gamma$ and $P'$;
3. $P = X(c_1, \ldots, c_n)$ implies:
   (i) $\Theta \vdash X; S_1, \ldots, S_n$;
   (ii) $\Gamma = \Gamma_0, \Gamma_1, \ldots, \Gamma_n$;
   (iii) $\text{end}(\Gamma_0)$;
4. $P = (\forall s:G) P$ implies:
   (i) $s \not\in \Gamma$;
   (ii) $\Theta \cdot \Gamma, \Gamma' \vdash P$, for some $\Gamma'$ such that $\Gamma' = \{ s[p]: G[p] \}_{p \in G}$;
5. $P = P_1 | P_2$ implies:
   (i) $\Gamma = \Gamma_1, \Gamma_2$ such that $\Theta \cdot \Gamma_1 \vdash P_1$ and $\Theta \cdot \Gamma_2 \vdash P_2$;
6. $P = c[\sum_{i \in I} m_i(y_i)].P_1$ implies:
   (i) $\Gamma = \Gamma_0, \Gamma_1$ such that $\Theta \vdash \Gamma_0, y_i; S_i, S'_i$ and $\Gamma_1 \vdash c[\sum_{i \in I} m_i(y_i)].P_1$;
7. $P = c[\sum_{i \in I} m_i(y_i)].P'$ implies:
   (i) $\Gamma = \Gamma_0, \Gamma_1, \Gamma_2$ such that $\Theta \vdash \Gamma_0, y_i; S_i, S'_i$ and $\Gamma_1 \vdash c[\sum_{i \in I} m_i(y_i)].P_1$;
   (ii) $\Gamma_2 \vdash d:S$ and $\Theta \cdot \Gamma_0, c:S' \vdash P'$.

Proof. Straightforward by the rules in Fig. 2, noticing that they are syntax-driven: i.e., for each shape of $P$ in cases 1–7 of the statement, the typing judgement in the hypothesis can be obtained by exactly one rule. More in detail, we have:

- case 1 by rule $[T\cdot\Theta]$;
- case 2 by rule $[T\cdot\text{def}]$;
- case 3 by rule $[T\cdot X]$;
- case 4 by rule $[T\cdot\text{CLASSIC}]$;
- case 5 by rule $[T\cdot I]$;
- case 6 by rule $[T\cdot &]$;
- case 7 by rule $[T\cdot \&]$.

Proposition I.5. Assume $\Theta \cdot \Gamma \vdash P$. Then, $\text{fc}(P) \subseteq \text{dom}(\Gamma)$ and $\forall c \in (\text{dom}(\Gamma) \setminus \text{fc}(P)) : \Gamma(c) \leq \text{end}$.

Proof. By induction on the typing derivation of $\Theta \cdot \Gamma \vdash P$.

Proposition I.6. Assume $\Theta \cdot \Gamma \vdash P$. Then, $\text{fpv}(P) \subseteq \text{dom}(\Theta)$.
We now prove the two claims. 

**Proposition I.7.** Assume \( \Theta \cdot \Gamma \vdash P \) and \( \Theta \cdot \Gamma' \vdash P \). Then:

1. \( \forall c \in \text{dom}(\Gamma) \cap \text{dom}(\Gamma') : \Gamma(c) \subseteq \Gamma'(c) \) or \( \Gamma'(c) \subseteq \Gamma(c) \);
2. \( \forall c \in \text{dom}(\Gamma) \setminus \text{dom}(\Gamma') : \Gamma(c) \subseteq \text{end} \);
3. \( \forall c \in \text{dom}(\Gamma') \setminus \text{dom}(\Gamma) : \Gamma'(c) \subseteq \text{end} \).

**Proof.** By induction on the typing derivation of \( \Theta \cdot \Gamma \vdash P \), observing the shape of \( P \) and the consequent constraints on the entries of \( \Gamma' \) imposed by Lemma I.4.

**Proposition I.8.** Assume \( \Theta \cdot \Gamma \vdash P \) and \( \Theta \cdot \Gamma' \vdash P \). Further, assume that \( P \) has guarded definitions, by \( \Gamma \). Then, \( P \) has guarded definitions, by \( \Gamma' \).

**Proof.** By induction on the typing derivation of \( \Theta \cdot \Gamma \vdash P \), applying Prop.1.7 to determine the shape of any alternative typing context \( \Gamma' \). The key observation is that \( \Gamma \) and \( \Gamma' \) do not influence the types of the bound variables \( x_1, \ldots, x_n \) in Def. 5.3(1).

**Proposition I.9.** Assume \( \Theta \cdot \Gamma \vdash P \) and \( P \equiv P' \). Further, assume that \( P \) has guarded definitions, by \( \Gamma \). Then, \( P' \) has guarded definitions, by \( \Gamma \).

**Proof.** By cases on the definition of \( \equiv \), and by applying Def. 5.3(1) — noticing that by Lemma B.2, we have \( \Theta \cdot \Gamma \vdash P' \).

**Lemma I.10.** Assume \( \Theta \cdot \Gamma \vdash P \) and \( \Theta \cdot \Gamma' \vdash P \). Further, assume that \( P \) only plays role \( p \) in session \( s \), by \( \Gamma \). Then, \( P \) only plays role \( p \) in session \( s \), by \( \Gamma' \).

**Proof.** Follows by Prop.I.8, Prop.I.7 and Def. 5.3.

**Proposition I.11.** If \( P \equiv 0 \), then \( \Theta \cdot \Gamma \vdash P \) implies \( \text{end}(\Gamma) \).

**Proof.** Assume \( \Theta \cdot \Gamma \vdash P \) with \( P \equiv 0 \). By Lemma B.2, \( \Theta \cdot \Gamma \vdash 0 \); hence, by Lemma I.4(1), we conclude \( \text{end}(\Gamma) \).

**Lemma I.12.** Assume \( \Theta \cdot \Gamma \vdash P \) and \( \Gamma = \Gamma_0, c: S \) with \( \text{end}(\Gamma_0) \) and \( q \&_{j \in J} m_j(S'_j) \subseteq S \). Further, assume that for each subterm \((v_s' : \Gamma') P' \) of \( P \), we have \( \text{end}(\Gamma') \). Then, \( P \equiv \text{def } D \text{ in } (v_s) P_1 \ldots | P_n \), where:

1. there is exactly one \( j \in 1..n \), such that either:
   a. \( P_j = \text{c[a]} @ m_k(d) P'_j \) for some \( k \in J \), or
   b. \( P_j = X(d_1, \ldots, d_{l-1}, c, d_{l+1}, \ldots, d_m) \) for some \( l, m \) such that \( 1 \leq l \leq m \);
2. for all \( i \in 1..n \) and \( i \neq j \) (with \( j \) from item 1 above), \( P_i = X(d_1, \ldots, d_m) \) for some \( m \).

**Proof.** (Sketch) The congruence for \( P \) holds by Prop.I.2, from which we also get:

\[
\forall i \in 1..n : P_i \text{ is either a branching, a selection, or a process call (by Prop.I.2)} \tag{21}
\]

We now prove the two claims.

**Item 1.** By Lemma I.4(5), \( c: S \) can only appear in the typing context \( \Gamma_j \) of exactly one \( P_j \), for some \( j \in 1..n \); moreover, the rest of the entries of \( \Gamma_j \) must be (subtypes of) \( \text{end} \), by the hypothesis on the typing of restricted sessions, and \( [T-\text{end}] \). Therefore, \( P_j \) cannot be a branching, by the contrapositive of Lemma I.4(6). Thus, by (21) and by the rules in Fig.2, we conclude that the only possible typable shapes for \( P_j \) are either (a) (by rule \( [T-\text{end}] \)) or (b) (by rule \( [T-X] \)). This proves the thesis.

**Item 2.** Observe that for all \( i \in 1..n \) and \( i \neq j \) (with \( j \) from item 1 above), we have \( \Theta_j : \Gamma_j \vdash P_i \) with \( \text{end}(\Gamma_j) \). By the contrapositive of Lemma I.4(6), \( P_i \) cannot be a branching; moreover, by the contrapositive of Lemma I.4(7), \( P_i \) cannot be a selection. Thus, by (21) and by the rules in Fig.2, we conclude that the only possible typable shape for \( P_i \) is \( P_i = X(d_1, \ldots, d_m) \) (by rule \( [T-X] \)). This proves the thesis.

**Lemma I.13.** Assume \( \Theta \cdot \Gamma \vdash P \) and \( \Gamma = \Gamma_0, c: S \) with \( \text{end}(\Gamma_0) \) and \( q \&_{j \in J} m_j(S'_j) \subseteq S \). Further, assume that for each subterm \((v_s' : \Gamma') P' \) of \( P \), we have \( \text{end}(\Gamma') \). Then, \( P \equiv \text{def } D \text{ in } (v_s) P_1 \ldots | P_n \), where:

1. there is exactly one \( j \in 1..n \), such that either:
   a. \( P_j = \text{c[a]} \sum_{i \in K} m_i(d) P'_j \) for some \( K \supseteq J \), or
   b. \( P_j = X(d_1, \ldots, d_{l-1}, c, d_{l+1}, \ldots, d_m) \) for some \( l, m \) such that \( 1 \leq l \leq m \);
2. for all \( i \in 1..n \) and \( i \neq j \) (with \( j \) from item 1 above), \( P_i = X(d_1, \ldots, d_m) \) for some \( m \).

We now prove the two claims.
We observe:

\[ \forall i \in 1..n : P_i \text{ is either a branching, a selection, or a process call} \quad \text{(by Prop.1.2)} \]  

(22)

We now prove the two claims.

**Item 1.** By Lemma I.4(5), \( c:S \) can only appear in the typing context \( \Gamma_j \) of exactly one \( P_j \), for some \( j \in 1..n \); moreover, the rest of the entries of \( \Gamma_j \) must be (subtypes of) \( \text{end} \), by the hypothesis on the typing of restricted sessions, and \( \text{[T-end]} \). Therefore, \( P_i \) cannot be a selection, by the contrapositive of Lemma I.4(7). Thus, by (22) and by the rules in Fig.2, we conclude that the only possible typable shapes for \( P_j \) are either (a) (by rule \([\text{T-\&}]\)) or (b) (by rule \([\text{T-\\&}]\)). This proves the thesis.

**Item 2.** Observe that for all \( i \in 1..n \) and \( j \neq j \) (with \( j \) from item 1 above), we have \( \emptyset \cdot \Gamma_j \vdash P_i \) with \( \text{end} (\Gamma_j) \). By the contrapositive of Lemma I.4(7), \( P_i \) cannot be a selection; moreover, by the contrapositive of Lemma I.4(6), \( P_i \) cannot be a branching. Thus, by (22) and by the rules in Fig.2, we conclude that the only possible typable shape for \( P_i \) is \( X_1(d_1, \ldots, d_m) \) (by rule \([\text{T-X}]\)). This proves the thesis.

\[ \square \]

**Theorem B.4 (Session Inversion).** Assume \( \emptyset \cdot \Gamma \vdash \| p \in I \| p \in P \) with each \( P_p \) either being 0 (up-to \( \equiv \)), or only playing role \( p \) in \( s \). Then, \( \Gamma = \Gamma_0, \{ s[p] : S_p \}_p \in I' \) (for some \( I' \)) with \( \text{end} (\Gamma_0) \). Moreover, \( \forall p \in I' : \)

1. if \( q \& j \in \text{maj}(S'_j), S'_j \subseteq S_p \) then \( p \in I \) and for some \( C, C', k \in J \), either:
   a. \( P_p \equiv \{ s[p] \equiv_{\sum_k m_k (x_k) \cdot P'_k} \} \) or
   b. \( P_p \equiv \{ \) def \( X_1(T_1, \ldots, x_n \cdot T_n) = C'[x_1[q] \equiv m_k (d) \cdot P'_p] \) in \( \langle \sum_k m_k (x_k) \cdot P'_k \rangle \) with \( 1 \leq l \leq n; \)

2. if \( q \& j \in \text{maj}(S'_j), S'_j \subseteq S_p \) then \( p \in I \) and for some \( C, C', k \in J \), either:
   a. \( P_p \equiv \{ s[p] \equiv_{\sum_k m_k (x_k) \cdot P'_k} \} \) or
   b. \( P_p \equiv \{ \) def \( X_1(T_1, \ldots, x_n \cdot T_n) = C'[x_1[q] \equiv m_k (d) \cdot P'_p] \) in \( \langle \sum_k m_k (x_k) \cdot P'_k \rangle \) with \( 1 \leq l \leq n; \)

3. if \( \text{end} \subseteq S_p \) then \( p \in I \) implies \( P_p \equiv 0 \).

Further, \( \forall p \in I \setminus I' : P_p \equiv 0 \).

**Proof.** Assume the hypotheses:

\[ \exists ! \alpha : \emptyset \cdot \Gamma \vdash \| p \in I \| p \in P \]  

\[ \forall p \in I : \begin{array}{l}
\text{ either:} \\
\text{ } P_p \equiv 0 \quad \text{or} \\
\text{ } P_p \text{ only plays role } p \text{ in } s \text{ (Def. 5.3)}
\end{array} \]

(23) \hspace{1cm} (24) \hspace{1cm} (25) \hspace{1cm} (26)

We observe:

\[ \Gamma = \Gamma_0, \ldots, \Gamma_n \quad \text{such that} \]

\[ I = \{ p_1, \ldots, p_n \} \quad \text{and} \quad \forall p \in I : \emptyset \cdot \Gamma_p \vdash P_p \quad \text{by (23) and Lemma I.4(5)} \]

(27)

if \( P_p \) only plays \( p \) in \( s \), then \( s[p] \in \text{fc}(P_p) \) and \( \text{fv}(P_p) = \emptyset \) (by Lemma I.10, Def. 5.3.3. and Prop.1.5, and (23))

(28)

Notice that:

\[ \exists ! \Gamma' : \Gamma_0, \{ s[p] : S_p \}_{p \in I'} \quad \text{and} \quad \text{end} (\Gamma_0) \]  

(29)

and (29) satisfies the first claim in the statement. We now prove the remaining claims.

**Item 1.** Take any \( p \in I' \) such that \( q \& j \in \text{maj}(S'_j), S'_j \subseteq S_p \). We first prove that \( p \in I \). By contradiction, assume \( p \notin I \) (i.e., none of the processes plays only role \( p \) in \( s \)); then, by (25), (26) and (28), we obtain that \( s[p] \notin \text{fc}(\| p \in I \| p \in P \) which (by Prop.1.5) implies \( S_p \neq \text{end} \). But then, by Def. 2.5, we obtain \( q \& j \in \text{maj}(S'_j), S'_j \neq S_p \) — contradiction. Hence, we have proved \( p \in I \). We now examine the two possible cases for \( P_p \):

1. \( P_p \equiv 0 \) (by (25)). We show that this case is absurd. By contradiction, assume \( P_p \equiv 0 \). Observe that \( \forall x \in I \) such that \( r \neq p, s[p] \notin \text{fc}(r) \) (by (25), (26) and (28)), which (by Def. 2.5) implies \( S_p \neq \text{end} \). But then, by Def. 2.5, we obtain \( q \& j \in \text{maj}(S'_j), S'_j \neq S_p \) — contradiction. Hence, we have proved \( P_p \equiv 0 \);
• \( P_p \) only plays role \( p \) in \( s \) by (26). Then, we apply Lemma L12, and we have two sub-cases. If Lemma L12(1)(a) holds, we observe that \( d \) therein cannot be a variable (otherwise we would have \( \text{fv}(P_p) \neq \emptyset \), i.e., by Def. 5.3, \( P_p \) does not only play \( p \) in \( s \) – contradiction), and therefore \( d = s'[r] \) (for some \( s', r \)), from which we conclude by obtaining case (a) of the statement with the following context:

\[
\mathcal{C} = \text{def } \overline{D} \text{ in } (v5) \mid Q_2 \mid \ldots \mid Q_n
\]  

(30)

Otherwise, if Lemma L12(1)(b) holds, we observe:

(1) by (23) and Prop.1.6, \( X \not\in \text{fpv}(P_p) \), and therefore, by Prop.1.2, \( P_p \) has the form:

\[
P_p = \text{def } \overline{D_1} \text{ in } (v3) X(x_1:T_1, \ldots, x_m:T_m) = Q \quad \text{in } \text{def } \overline{D_2} \text{ in } (v5) X(d_1, \ldots, d_n)
\]  

(31)

(2) hence, we can use the congruence \( \equiv \) to remove unused process declarations \( \overline{D_2} \) in (31), and place the definition of \( X \) immediately before the call:

\[
P_p = \text{def } \overline{D_1} \text{ in } (v5) \text{ def } X(x_1:T_1, \ldots, x_m:T_m) = Q \quad \text{in } X(d_1, \ldots, d_n)
\]  

(32)

(3) from (32), by Lemma L1.4 and Lemma B.2, and observing that \( d_1, \ldots, d_n \) cannot be variables (since \( \text{fv}(P_p) = \emptyset \), by Def. 5.3 and Prop.1.1), we get \( \Gamma_p = \Gamma_{p0}, \ldots, \Gamma_{pm} \) such that (note that the following derivation applies \( \vdash \text{def} \) once for each process declaration in \( \overline{D_1} \), yielding the sequence of process typings \( X':T' \)):

\[
\begin{align*}
\Theta, X':T', X:T_1, \ldots, T_n + X:T_1, \ldots, T_m \quad \forall i \in 1..n \quad \Gamma_{pi} + s_i'[r_i]:T_i \\
X':T'. X:T_1, \ldots, T_n + \text{def } \Gamma_{pi} + X\langle s_i'[r_i], \ldots, s_n'[r_n] \rangle
\end{align*}
\]  

(33)

From (33), we get:

\[
\begin{align*}
m = n & \quad \text{(34)} \\
\exists l: s_l'[r_l] = s[p] & \quad \text{(35)} \\
\forall i \in 1..n: i \neq l \text{ implies } \text{end } \leq T_i & \quad \text{(by Def. 5.3(2)(iv)} \\
\forall q \in \text{def } q(p,S_i). S_i' \subseteq S_p \leq T_l & \quad \text{(36)}
\end{align*}
\]  

(37)

(4) by Prop.1.2 we have:

\[
Q \equiv \text{def } \overline{D''} \text{ in } (v5) Q_1 | \ldots | Q_n
\]  

(38)

where \( \forall i \in 1..n'' : Q_i \) is either a branching, a selection, or a process call

\[
X':T'. X_i:T_1, \ldots, x_i:T_i, \ldots, x_m:T_m + Q \quad \text{(by (33))}
\]  

(39)

(5) since \( P_p \) has guarded definitions (by Def. 5.3(2), then by (38), Prop.1.9 and (39) we know that in \( Q_1 | \ldots | Q_n \) (from (38)) some branching or selection uses \( x_i \), before further process calls. Without loss of generality, assume that \( Q_1 \) satisfies the requirement;

(6) by (39) and the contrapositive of Lemma 1.4(6), \( Q_1 \) cannot use \( x_i \) for branching;

(7) hence, by rule \( \vdash \text{end} \), we conclude that \( Q_1 \) is a selection on \( x_i \);

(8) therefore, by Lemma L12(1)(a), we obtain:

\[
Q_1 = x_i[q] \oplus m_k(d). P'_p \quad \text{for some } k \in j
\]  

(40)

Summing up, from (32), (34), (35), (38) and (40) we have the following reduction contexts, as required by case (b) of the statement:

\[
\begin{align*}
\mathcal{C} & = \text{def } \overline{D_1} \text{ in } (v5) \left( \text{def } X(x_1:T_1, \ldots, x_n:T_n) = C' \text{ in } X(s_1'[r_1], \ldots, s_{n-1}'[r_{n-1}], s[p], s_{i+1}'[r_{i+1}], \ldots, s_n'[r_n]) \right) \\
C' & = \text{def } \overline{D''} \text{ in } (v5) [ ] | Q_2 | \ldots | Q_n
\end{align*}
\]  

(41)

and this concludes the proof.
Item (2). The proof is similar to that for item (2) above, except that we use Lemma I.13 instead of Lemma I.12, obtaining the same reduction contexts: either (30) (thus obtaining case (a) of the statement) or (41) (thus obtaining case (b) of the statement).

Item (3). If \( I \cap I' = \emptyset \), the statement holds vacuously. Otherwise, take any \( p \in I \cap I' \), and assume \( \text{end} \leq S_p = \Gamma(s[p]) \).

We have two cases:
- \( P_p = \emptyset \) (by (25)). This is the thesis;
- \( P_p \) only plays role \( p \) in \( s \) (by (26)). This case is impossible: otherwise, by (28) we would get \( S_p \not\leq \text{end} \), and thus \( \text{end} \not\leq S_p \) (by Prop.I.13) — which would contradict the assumption \( \text{end} \leq S_p \).

Item (4). If \( I \setminus I' = \emptyset \), the statement holds vacuously. Otherwise, take any \( p \in I \setminus I' \), which implies \( p \not\in I' \). We have two cases:
- \( P_p = \emptyset \) (by (25)). This is the thesis;
- \( P_p \) only plays role \( p \) in \( s \) (by (26)). This case is impossible: otherwise, by (28) we would get \( \Gamma_p = \Gamma'_p \), \( s[p]:S_p \) and therefore \( s[p] \in \Gamma(\Gamma) \) (by (27)), that means \( p \in I' \) (by (29)) — which would contradict the assumption \( p \not\in I' \).

\[ \square \]

Definition I.14 (Type Unfolding). The one-step unfolding of a type \( S \), written \( \text{unf}(S) \), is:

\[ \text{unf}(\mu t.T) = T[\mu t.T/t] \quad \text{unf}(T) = T \quad \text{if} \quad T \not\equiv \mu t.T' \]

The \( n \)-steps unfolding of a type \( S \), written \( \text{unf}^n(S) \), is:

\[ \text{unf}^0(T) = T \quad \text{unf}^{m+1}(T) = \text{unf}(\text{unf}^m(T)) \]

The complete unfolding of a session type \( S \), written \( \text{unf}^*(S) \), is defined as:

\[ \text{unf}^*(S) = \text{unf}^n(S) \quad \text{for the smallest} \quad n \text{ such that} \quad \text{unf}^n(S) = \text{unf}^{n+1}(S) \]

Proposition I.15. For all \( S, T \):

1. \( S \leq T \quad \text{if and only if} \quad \text{unf}(S) \leq T \), and
2. \( S \leq T \quad \text{if and only if} \quad S \leq \text{unf}(T) \).

Proof. Item 1.

- (\( \Rightarrow \)) Assume \( S \leq T \). If \( S \not\equiv \mu t.S' \), then \( S = \text{unf}(S) \leq T \), and we obtain the thesis. Otherwise, by Def. I.14, we have \( \text{unf}(S) = S'[\mu t.S'/t] \), and we conclude by the coinductive rule [\text{Sub-\( \mu \)L}] of Def. 2.5.

- (\( \Leftarrow \)) Assume \( \text{unf}(S) \leq T \). If \( S \not\equiv \mu t.S' \), then \( \text{unf}(S) = S \leq T \), and we obtain the thesis. Otherwise, by Def. I.14, we have \( S'[\mu t.S'/t] \leq T \), and since \( \leq \) is the largest relation closed backward under the coinductive rule [\text{Sub-\( \mu \)L}] of Def. 2.5, we conclude \( \mu t.S' = S \leq T \).

Item 2. The proofs for the two the implications in the statement are similar to the corresponding proofs for item 1, but using the coinductive rule [\text{Sub-\( \mu \)R}] of Def. 2.5 (instead of [\text{Sub-\( \mu \)L}]).

\[ \square \]

Proposition I.16. For all \( S, T \):

1. \( S \leq T \quad \text{if and only if} \quad \text{unf}^*(S) \leq T \), and
2. \( S \leq T \quad \text{if and only if} \quad S \leq \text{unf}^*(T) \).

Proof. Take the smallest \( n \) such that \( \text{unf}^*(S) = \text{unf}^n(S) \) (by Def. I.14).

- Item 1 By induction on \( n \), applying Prop.I.15(1) in the inductive case.
- Item 2 By induction on \( n \), applying Prop.I.15(2) in the inductive case.

\[ \square \]

Proposition I.17. \( s[p]:S, s[q]:T \rightarrow \Gamma' \quad \text{if and only if} \quad s[p]:\text{unf}(S), s[q]:T \rightarrow \Gamma' \).

Proof. (\( \Rightarrow \)). If \( S \not\equiv \mu t.S' \), then \( S = \text{unf}(S) \), and the statement holds vacuously. Otherwise, we conclude by Def. I.14 and Def. 2.8 (rule [\text{\Gamma-COMM}]).

(\( \Leftarrow \)). If \( S \not\equiv \mu t.S' \), then \( S = \text{unf}(S) \), and the statement holds vacuously. Otherwise, we conclude by Def. I.14 and inversion of rule [\text{\Gamma-COMM}] in Def. 2.8.

\[ \square \]

Proposition I.18. \( s[p]:S, s[q]:T \rightarrow \Gamma' \quad \text{if and only if} \quad s[p]:\text{unf}^*(S), s[q]:T \rightarrow \Gamma' \).

Proof. Take the smallest \( n \) such that \( \text{unf}^*(S) = \text{unf}^n(S) \) (by Def. I.14). The statement is proved by induction on \( n \), applying Prop.I.17 in the inductive case.

\[ \square \]
**Lemma I.19.** Assume \( s[p]; s[q] : T \rightarrow \). Moreover, assume:

\[
\Gamma = \Gamma_0, s[p]; s[q] : T \leq \Gamma' = \Gamma_0', s[p]; s[q] : T' \rightarrow \Gamma'' = \Gamma_0'', s[p]; s[q] : T''
\]

Then, there is \( \Gamma''' \) such that \( \Gamma \rightarrow \Gamma''' \leq \Gamma'' \).

**Proof.** Assume:

\[
s[p]; s[q] : T \rightarrow \Gamma \leq \Gamma'
\]

\[
\Gamma' \rightarrow \Gamma''
\]

We have:

\[
\begin{align*}
\Gamma' &= \Gamma_0', s[p]; s[q] : T' \\
&\text{with } \text{unf}'(S') = q \otimes_{i \in J} m_i(S_i'), S_i''', \\
&\text{and } \text{unf}'(T') = p \otimes_{j \in J} m_j(T_j'), T_j'''
\end{align*}
\]

(by (44), Def. 2.8, Prop. I.18)

\[
\Gamma'' = \Gamma_0'', s[p]; s[q] : T''
\]

(by (44), (45), Prop. I.18)

\[
\Gamma = \Gamma_0, s[p]; s[q] : T
\]

(by (43), (45), Prop. I.16, Def. 2.5)

We conclude:

\[
\begin{align*}
\Gamma' \rightarrow \Gamma'''
\end{align*}
\]

(by (47), (49), (48), Def. 2.8, Prop. I.18)

\[
\Gamma''' \leq \Gamma''
\]

(by (47), (49), (48), Def. 2.6)

\[ \square \]

**Lemma B.3 (Narrowing).** If \( \Theta \cdot \Gamma \vdash P \) and \( \Gamma \leq \Gamma' \), then \( \Theta \cdot \Gamma' \vdash P \).

**Proof.** By induction on the derivation of \( \Theta \cdot \Gamma \vdash P \), and by Def. 2.5. \[ \square \]

**Theorem 5.4 (Session Fidelity).** Assume \( \Theta \cdot \Gamma \vdash P \), where \( \Gamma \) is safe, \( P \equiv |_{\Gamma \vdash P} \), and each \( P_m \) either is 0 (up-to \( \equiv \)), or only plays \( p \) in \( s \). Then, \( \Gamma \vdash P \) implies \( \exists \Gamma', P' \) such that \( \Gamma \rightarrow \Gamma' \), \( P \rightarrow P' \) and \( \Theta \cdot \Gamma' \vdash P' \), where \( P' \equiv |_{\Gamma \vdash P'} \), and each \( P_m' \) either is 0 (up-to \( \equiv \)), or only plays \( p \) in \( s \).

**Proof.** We have:

\[
\begin{align*}
\text{safe}(\Gamma) \\
\Gamma \rightarrow \\
\Gamma \leq \Gamma_0, s[p]; s[q] : T
\end{align*}
\]

(by hypothesis)

(by hypothesis)

(by (52), (53), Def. 2.8, Prop. I.18, Prop. I.16)

\[ \square \]
We develop the proof for case (b) (the other cases are similar).
By induction on \(C\) and \(C_p\) using Lemma I.4, we can prove that the term inside \(C_p\) is typed as:

\[
\begin{align*}
\Theta_p, \Theta_p', Y: T_1, \ldots, T_m, & \quad \Gamma_0, \Gamma_0', y' : \Delta_k \vdash \Gamma_{pd'} + d : T_k'' \\
\Theta_p, \Theta_p', Y: T_1, \ldots, T_m, & \quad \Gamma_0, y' : \Delta_k \vdash \Gamma_{pd'} + T_p'' \\
\vdots
\end{align*}
\]

The term above can reduce by rule [\(\R\)-X] (Fig. 1), becoming the term (56) below, that we can type from (55) by applying Lemma B.1 \(m\) times (one per argument of \(Y\)). Notice, in particular, that in (55) \(\Delta_k\) can come either from the call substitutions (i.e., it replaces some \(y_j, j' \in \Gamma_{pd}\), or from the restriction in \(C_p\); in both cases, \(s''[r''']\) is typed by some \(\Gamma_{ps'''}\) (taking the place of \(\Gamma_{pd}''\) in (55)):

\[
\Gamma_{pd'} + s[p] : \Delta_k \\
\forall i \in 1..(i'' - 1), (i'' + 1)..m \\
\Gamma_{pd'} + s_j'[r_j] : T_i \\
\vdots
\]

Applying the above reduction to the process in case (b) (via rule [\(\R\)-Crx]), and rearranging the reduction contexts via \(\equiv\) into a single context \(C_{pq}\) with one hole (using Prop. I.2), we get the following reduction, by (56) and [\(\R\)-\(=\)]:

\[
P \to P'' \equiv C_{pq} \quad \begin{aligned}
& s[p][q] \equiv m_k \langle s''[r'''] \rangle. P'_{pq} \langle s_i'[r_i] / y_i \rangle \cdots \langle s_i'[r_i] / y_m \rangle \\
& \quad \mid s[q][p] \sum_{i \in L} m_i(X_i). P'_{ql}\end{aligned}
\]

Notice that the typing context in the conclusion of (56) is the same of (55) (since the reduction involves a closed process variable \(Y\) and does not use any channel). Therefore:

\[
0 \cdot \Gamma + C_{pq} \mid s[p][q] \equiv m_k \langle s''[r'''] \rangle. P'_{pq} \langle s_i'[r_i] / y_i \rangle \cdots \langle s_i'[r_i] / y_m \rangle \\
\mid s[q][p] \sum_{i \in L} m_i(X_i). P'_{ql}
\]

(by the hyp. \(0 \cdot \Gamma + P\), (57) and Lemma B.2) (58)
By (56) and Lemma 1.4, the term inside $C_{pq}$ is typed as:

$$
\Gamma_p' \vdash s[p] : q \qeq \sum_{k} (T''')_{k} \quad \Theta_{p'}' \vdash \sum_{k} (T''')_{k} \quad \Gamma_{ps''} \vdash s''[r'][:T''']
$$

By (59), $[T-\Sigma]$ and (61), and the rest of the hypotheses in the statement hold.

$$
\Theta_{p'}' \vdash \sum_{k} s[p][q] \qeq \sum_{k} (s''[r'][:T'''])_{k} \quad \Gamma_{ps''} \vdash s''[r'][:T''']
$$

and thus $\Gamma_{ps''} \vdash s''[r'][:T''']$ (by (59), $[T-\Sigma]$ and transit. of $\leq$) (60)

The process in (59) reduces to the following process (by $[R-\Sigma\text{Comm}]$), which we can type by (60) and Lemma B.1:

$$
\begin{align*}
\Theta_p' \cdot \Gamma_{ps''} \cdot \Gamma_{p'} \cdot \Gamma_{q_0} \cdot \Gamma_q \vdash P_{0} \& \sum_{k} s[p][q] \qeq \sum_{k} (U''')_{k} \cdot U'''

\Theta_{p'}' \cdot \Gamma_{p_0} \cdot \Gamma_{p'} \cdot \Gamma_{q_0} \cdot \Gamma_{q} \vdash P'_{0} \cdot \sum_{k} s[p][q] \qeq \sum_{k} (U''')_{k} \cdot U'''

\forall \Gamma_{pq} \in [T-\Sigma] \quad \Theta_{p} \cdot \Gamma_{ps'} \vdash P''_{p} \cdot \sum_{k} s'[r]/y_{k} \cdot \sum_{k} s'[r_{m}]/y_{m} \quad \Theta_{q} \cdot \Gamma_{qs'} \vdash P''_{q} \cdot \sum_{k} s''[r'/x_{k}]

\end{align*}
$$

Notice that the process typing context $\Theta_{p} \cdot \Theta_{q}$ does not change in the reduction from (59) to (61). For the channel typing context, instead, we have:

$$
\begin{align*}
\Gamma_{p_0}'' \cdot \Gamma_{ps''}'' \cdot \Gamma_{p'}'' \cdot \Gamma_{q_0}'' \cdot \Gamma_{q}'' \vdash \sum_{k} s[p][q] \qeq \sum_{k} (U''')_{k} \cdot U'''

\Gamma_{pq}'' \vdash \sum_{k} s[p][q] \qeq \sum_{k} (U''')_{k} \cdot U'''

\forall \Gamma_{pq}'' \in [T-\Sigma] \quad \Theta_{p}'' \cdot \Gamma_{ps'}'' \vdash P''_{p} \cdot \sum_{k} s'[r]/y_{k} \cdot \sum_{k} s'[r_{m}]/y_{m} \quad \Theta_{q}'' \cdot \Gamma_{qs'}'' \vdash P''_{q} \cdot \sum_{k} s''[r'/x_{k}]

\end{align*}
$$

Observe that in (62) and (63), the entries for $s[p]$ and $s[q]$ reduce, and (by (63) and Def. 2.8) such entries are the only difference between $\Gamma_{p_0}'' \cdot \Gamma_{ps''}'' \cdot \Gamma_{p'}'' \cdot \Gamma_{q_0}'' \cdot \Gamma_{q}''$ and $\Gamma_{pq}''$. By applying the same update of $s[p]$ and $s[q]$ to $\Gamma'$ from the statement, we obtain a typing context $\Gamma''$ such that:

$$
\begin{align*}
\emptyset \cdot \Gamma'' + P' & \equiv C_{pq} \left( P''_{p} \cdot \sum_{k} s'[r]/y_{k} \cdot \sum_{k} s'[r_{m}]/y_{m} \right) \quad P''_{q} \cdot \sum_{k} s''[r'/x_{k}]

\end{align*}
$$

This is proved by noticing that, from the hypothesis (b), (66) and $[R=\Sigma]$, the reductions from $P$ to $P'$ yield:

$$
\begin{align*}
P & \rightarrow P'' \\
P & \rightarrow P'' \\
& \rightarrow P'

\end{align*}
$$

Summing up, we have shown that if $\Gamma$ is safe and reduces ($\Gamma \rightarrow$), and the rest of the hypotheses in the statement hold, then there exist $\Gamma'$, $P''$ such that $\Gamma \rightarrow \Gamma'$ (by (67)), $P \rightarrow^* P''$ (by (66)) and $\emptyset \cdot \Gamma'' + P' \equiv P$ (by (65)).

We are left to prove that:

1. $P' \equiv \sum_{k} P'_{k}$. This is proved by noticing that, from the hypothesis (b), (66) and $[R=\Sigma]$, the reductions from $P$ to $P'$ yield:

$$
\begin{align*}
P & \rightarrow P'' \\
P & \rightarrow P'' \\
& \rightarrow P'

\end{align*}
$$

and this satisfies the requirement.

2. each $P'_{k}$ has guarded definitions and is either 0 (up-to $\equiv$), or only plays role $r$ in $s$. From (68), since $Q$ and the reduction contexts $C_{p}$, $C_{pq}$, $C_{q}$ are unchanged w.r.t. the hypothesis (b), we only need to show that the process inside $C_{p}$ (resp. $C_{q}$) is either 0 (up-to $\equiv$), or only plays role $p$ (resp. $q$) in $s$. We only examine the process inside $C_{q}$ (the reasoning for the other is similar). If $P''_{qk} \cdot s''[r'/x_{k}] \equiv 0$, we have $C_{q} \left( P''_{qk} \cdot s''[r'/x_{k}] \right) \equiv 0$, and we conclude easily.

Otherwise, observe that:

$$
\emptyset \cdot \Gamma_{q}'' + C_{q} \left( P''_{qk} \cdot s''[r'/x_{k}] \right)
$$

where $\Gamma_{q}''$ is a part of $\Gamma'$ (by (68) and Lemma 1.4(5)) where the only non-end-typed channel is $s[q]$. Moreover, from the initial hypotheses, in all subterms ($v[s''':T'''']$) $P'''$ of $C_{q}$ and $P''_{qk}$, we have end($T'''$), and all the process
Therefore, by (70), (71), Def. 2.8 and Prop.1.18, we conclude
\[ \Gamma \vdash \square \text{safe} \]
with the induction hypothesis. The most interesting case is the base case where structural induction on the reduction context \( C \) by Def. 2.6, \( \Gamma \), J SUBJECT REDUCTION \[ \rightarrow \]
Assume Lemma 4.4. If Theorem 4.8 (Subject Reduction).
Proof. By induction of the derivation of \( P \rightarrow P' \), and by applying the safety hypothesis and Prop.1.19. Then, we conclude by Lemma 1.19.

Lemma 4.3. If \( \Gamma, \Gamma' \) is safe, then \( \Gamma \) is safe.
Proof. By contradiction, assume that \( \Gamma \) is not safe. Then, by Def. 4.1 (clause [S⇌]), there is \( \Gamma'' \) such that \( \Gamma \rightarrow^* \Gamma'' \), and \( \Gamma'' \) violates clause [S⇌&] (possibly after applying [S⇌µ] to unfold its entries). But then, by Def. 2.8 (rule [Γ-Cons]), \( \Gamma, \Gamma' \rightarrow^* \Gamma'' \), and the latter is not safe. This means that \( \Gamma, \Gamma' \) violates clause [S⇌] of Def. 4.1, and therefore is not safe: contradiction. We conclude that \( \Gamma \) is safe.

Theorem 4.8 (Subject Reduction). Assume \( \Theta \cdot \Gamma + P \) and \( \Theta \) safe. Then, \( P \rightarrow P' \) implies \( \exists \Gamma' \) safe such that \( \Gamma \rightarrow \Gamma' \) and \( \Theta \cdot \Gamma' + P' \).
Proof. By induction of the derivation of \( P \rightarrow P' \), and when the reduction holds by rule [R-Crx], with a further structural induction on the reduction context \( C \). Most cases hold by inversion of the typing \( \Theta \cdot \Gamma + P \), and by applying the induction hypothesis. The most interesting case is the base case where \( P \rightarrow P' \) holds by [R-Comms]:

\[
P = \{ p \mid q \} \sum_{i \in I} \Theta_i(x_i).P_i | \{ q \mid p \} \sum_{i \in J} \Theta_i(x_i).Q_i \quad \text{(by inversion of [R-Comms])} \tag{72}
\]

\[
P' = \{ p \mid q \} \sum_{i \in I} \Theta_i(x_i).P_i | \{ q \mid p \} \sum_{i \in J} \Theta_i(x_i).Q_i \quad \text{(by inversion of [R-Comms])} \tag{72}
\]

\[
\Gamma = \Gamma_{\&}, \Gamma_{\&} \quad \text{such that} \quad \Gamma_1 + \{ s \mid q \} \sum_{i \in I} \Theta_i(x_i).P_i \quad \text{[T-\&]} \tag{73}
\]

\[
\Gamma_{\&} = \Gamma_{\&}, \Gamma_1 \quad \text{such that} \quad \Gamma_3 + \{ q \mid p \} \sum_{i \in J} \Theta_i(x_i).Q_i \quad \text{[T-\&]} \tag{74}
\]

\[
\Gamma_{\&} = \Gamma_{\&}, \Gamma_3, \Gamma_4 \quad \text{such that} \quad \Gamma_4 + \{ q \mid p \} \sum_{i \in J} \Theta_i(x_i).Q_i \quad \text{[T-\&]} \tag{75}
\]
Now, notice that:

\[
\Gamma = \Gamma_0, \Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \quad \text{(by (73), (74), and (75))} \\
\Gamma_1 = s(p):S \quad \text{with} \quad S \subseteq q \& \in I \mathcal{m}_1(S_i), S'_i \quad \text{(by (74) and Fig. 2, rule \text{T-Sub})} \\
\Gamma_4 = s(q):T \quad \text{with} \quad T \subseteq q \& \mathcal{m}_k(T_k), T'_k \quad \text{(by (75) and Fig. 2, rule \text{T-Sub})} \\
\Gamma \subseteq \Gamma'' = \Gamma_0, s(p):q \& \in I \mathcal{m}_1(S_i), S'_i, \Gamma_2, \Gamma_3, s(q):q \& \mathcal{m}_k(T_k), T'_k \quad \text{(by (76), (77), (78), and Def. 2.6)} \\
\text{safe}(\Gamma'') \quad \text{(since safe}(\Gamma)\text{, and by (79) and Lemma 4.5)} \\
k \in I \quad \text{and} \quad T_k \leq S_k \quad \text{(by (79), (80) and Def. 4.1, clause [S->&])} \\
\Gamma'' \rightarrow \Gamma''' = \Gamma_0, s(p):S'_k, \Gamma_2, \Gamma_3, s(q):T'_k \quad \text{(by (79), (81) and Def. 2.8)} \\
\text{safe}(\Gamma''') \quad \text{(by (80), (82) and Def. 4.1, clause [S->])} \\
\text{We can now use} \Gamma''' \text{to type} P': \\
\Theta \cdot \Gamma_0, x_k: S_k, s(p): S'_k + P_k \quad \text{(by (81), (75) and (74))} \\
\Gamma_3 + s' | r | : S_k \quad \text{(by (75) for} \Gamma_3 \vdash s' | r | : T_k), (81), \text{transitivity of} \leq, \text{and} \text{T-Sub}) \\
\Gamma_0, \Gamma_3, s(p): S'_k \quad \text{defined} \quad \text{(by (75), (74), and (73))} \\
\Theta \cdot \Gamma_0, \Gamma_3, s[p]: S'_k + P_k \{s'[r]/x_k\} \quad \text{(by (84), (85), (86), and Lemma B.1)} \\
\Theta \cdot \Gamma_0, \Gamma_3, s[p]: S'_k + P_k \{s'[r]/x_k\} \quad \Theta \cdot \Gamma_2, s[q]: T'_k \quad [T-\text{I}] \\
\Theta \cdot \Gamma''' + P' \quad \text{(by (87), (75), (83) and (72))} \\
\text{We conclude the proof by showing that there exists some} \Gamma' \text{that satisfies the statement:} \\
\exists \Gamma' : \Gamma \rightarrow \Gamma' \leq \Gamma''' \quad \text{(by (79), (82), and Lemma 4.4)} \\
\text{safe}(\Gamma') \quad \text{(by (89) and Def. 4.1, clause [S->])} \\
\Theta \cdot \Gamma' + P' \quad \text{(by (88), (89), and Lemma B.3)} \\
\square

\text{PROPOSITION J.2.} \text{ The multiparty session subtyping relation} \leq \text{ is decidable.} \\

\text{PROOF.} \text{ An algorithm for checking whether a pair of types} S, T \text{ belongs to} \leq \text{ can be obtained as a variation of the binary session subtyping algorithm of [Gay and Hole 2005, Fig. 11] -- which in turn is based on the subtyping algorithm for recursive types of [Pierce 2002, Fig. 21-4]. See [Ghilezan et al. 2018] for a detailed description of the algorithm, and the paper artifact [Scalas and Yoshida 2019] for an implementation.} \square

\text{THEOREM 4.11.} \text{ If} \phi \text{ is decidable, then} "\Theta \cdot \Gamma \vdash P \text{ with} \phi" \text{ is decidable.} \\

\text{PROOF.} \text{ An algorithm for deciding} \Theta \cdot \Gamma \vdash P \text{ can be straightforwardly obtained by inverting the typing rules in Fig. 2 and Def. 4.6, noticing that:} \\
(1) \text{all typing rules are deterministically invertible -- i.e., for each shape of} P, \text{ at most one rule can conclude} \Theta \cdot \Gamma \vdash P; \\
(2) \text{at each inversion, the typing contexts} \Theta, \Gamma \text{ and the type annotations in} P \text{ determine how to populate the typing context in the premises the rule -- with the exception of} [T-I], \text{that (in the worst case) might require to try all possible} \Gamma_1, \Gamma_2 \text{ such that} \Gamma_1, \Gamma_2 \vdash \Gamma; \\
(3) \text{each rule inversion yields premises with strictly smaller subterms of} P \text{ (thus, recursive type checking eventually terminates).} \\
\text{Moreover, since} \leq \text{ is decidable (by Prop. J.2), it is always decidable whether the judgement}[T-\text{Sub}] \text{ holds. Finally, notice that when inverting rule} [T-\text{Sub}] \text{(Def. 4.6), the check} \phi(\Gamma') \text{ is decidable by hypothesis.} \square

\text{K TYPING CONTEXT PROPERTIES} \\
\text{Global Type Projections as Behavioural Properties.} \text{ To develop the proofs below, we establish a link between syntactic projections and behavioural typing context properties: this is done in Def. K.4, using some tools from Def. K.1.} \\

\text{DEFINITION K.1 (Typing Context Unfoldings and Behavioural Set).} \text{ The set of unfoldings of} \Gamma, \text{ written} \text{unf}(\Gamma), \text{ is defined as follows (where} \Gamma(s/c) \text{ is a mapping update):} \\
\text{unf(} \Gamma \text{)} = \bigcup_{c:S \in \Gamma} \{ \Gamma(s/c) \} \quad \text{extended to sets as} \quad \text{unf(} \Gamma_i \text{)} = \bigcup_{i \in I} \text{unf}(\Gamma_i)
Given a set of typing contexts $E$, the closure of the unfoldings of its elements is:

$$\text{unf}^*(E) = \text{fix} \left( \lambda E'. \ E \cup E' \cup \text{unf}(E \cup E') \right)$$

where $\text{fix}$ is the least fixed point of its argument.

Given $\Gamma$, the behavioural set of $\Gamma$, written $\text{beh}(\Gamma)$, is the set: $\text{beh}(\Gamma) = \text{unf}^* \left( \{ \Gamma' \mid \Gamma \rightarrow \Gamma' \} \right)$

In Def. K.1, $\text{beh}(\Gamma)$ is the set of all reductions of $\Gamma$, extended with all unfoldings of all their entries. This ensures that $\text{beh}(\Gamma)$ mechanically satisfies clauses $[S \rightarrow \sigma]$ and $[S \rightarrow \mu]$ of Def. 4.1 and Fig. 5 (cf. K.2 below). Therefore, one can verify whether $\text{beh}(\Gamma)$ is a safety/liveness property by only checking whether the remaining clauses hold for all elements — and if they do, it means that $\Gamma$ is safe/live. Note that the least fixed point in $\text{unf}^*(E)$ exists because the function is monotonic w.r.t. $\subseteq$ [Tarski 1955].

**Proposition K.2.** Let $\varphi = \text{beh}(\Gamma)$, for some $\Gamma$. Then, each element of $\varphi$ satisfies clauses $[S \rightarrow \sigma]$ and $[S \rightarrow \mu]$ of Def. 4.1.

For each safe $\Gamma$, Def. K.1 gives the smallest safety property, as formalised in Prop. K.3 below.

**Proposition K.3.** $\Gamma$ is safe (resp. live, live*) if and only if $\text{beh}(\Gamma)$ is a safety (resp. liveness, liveness*) property.

**Proof.** ($\implies$). Assume that $\Gamma$ is safe (resp. live, live*). We have to prove that each $\Gamma' \in \text{beh}(\Gamma)$ satisfies the clauses of Def. 4.1 (resp. 5(5), 5(6)). By Prop. K.2, we know that each $\Gamma'$ satisfies clauses $[S \rightarrow \sigma]$ and $[S \rightarrow \mu]$; the remaining clauses are easily proved by contradiction: if we assume that $\Gamma'$ does not satisfy some clause, by observing that $\Gamma'$ is (possibly unfolded) reduct of $\Gamma$, we obtain that $\Gamma$ is not safe (resp. live, live*) — contradiction. Therefore, each $\Gamma' \in \text{beh}(\varphi)$ satisfies all clauses of Def. 4.1 (resp. 5(5), 5(6)), and we conclude that $\text{beh}(\Gamma)$ is a safety (resp. liveness, liveness*) property.

($\impliedby$). Immediate by Def. 4.1 (resp. 5(5), 5(6)).

We can now use global type projections to produce behavioural properties that are directly usable in our framework. This is formalised in Def. K.4.

**Definition K.4.** With an abuse of notation, we will use the following definitions instead of those in Def. 5.8:

$$\text{fproj}_{G,s} = \text{beh}(\Gamma) \quad \text{where } \Gamma \text{ is the projection of } G \text{ for } s$$

$$\text{pproj}_{G,s} = \text{beh}(\Gamma) \quad \text{where } \Gamma \text{ is the plain projection of } G \text{ for } s$$

i.e., we extend the properties in Def. 5.8 to contain the unfoldings and reductions of a typing context $\Gamma$ projected from $G$. The results below hold for such extensions, and thus, also for the original Def. 5.8.

**Proposition K.5.** If consistent(\Gamma) and $\Gamma \rightarrow \Gamma'$, then consistent(\Gamma').

**Proof.** From [Scalas et al. 2017, Lemma D.22].

**Definition K.6 (Type contexts).** A global type context is defined as follows:

$$\mathcal{G} ::= \ p \rightarrow q \mid \{ m_i(S_i) \}_I \mid \mu t. \mathcal{G} \mid \lfloor I \rfloor$$

We write $\mathcal{G}[G_i]_{i \in I}$ to denote the global type obtained by filling the hole $\lfloor I \rfloor$ of $\mathcal{G}$ with the global type $G_i$, for all $i \in I$, with the understanding that $I$ indexes all the holes in $\mathcal{G}$.

A session type context is defined as follows:

$$\mathcal{S} ::= p \& I \{ m_i(T_i) \}_I \mid p \oplus I \{ m_i(T_i) \}_I \mid \mu t. \mathcal{S} \mid \lfloor I \rfloor$$

We write $\mathcal{S}[T_i]_{i \in I}$ to denote the session type obtained by filling the hole $\lfloor I \rfloor$ of $\mathcal{S}$ with the session type $T_i$, for all $i \in I$, with the understanding that $I$ indexes all the holes in $\mathcal{S}$.

We now adapt to our framework the notion of multiparty compatibility defined in [Deniélou and Yoshida 2013, Def. 4.2] and [Bocchi et al. 2015, Def. 4], for Communicating Finite-State Machines (CFSMs).

**Definition K.7 (Multiparty Compatibility [Bocchi et al. 2015; Deniélou and Yoshida 2013]).** Assume $\text{dom}(\Gamma) = \{ s \}$. We say that $\Gamma$ is multiparty compatible iff:

1. $\Gamma \rightarrow^s \Gamma', [p]:S$ with $\text{unf}^*(S) = q \& I \{ m_i(S_i) \}_I.S'_I$ implies $\exists k \in I: \exists \Gamma'', \Gamma''' \Gamma' \rightarrow^s \Gamma'' \text{ and } \Gamma'' \rightarrow^s \Gamma'''$, $[p]:S \rightarrow \Gamma''', [p]:S'_K$;

2. $\Gamma \rightarrow^s \Gamma', [p]:S$ with $\text{unf}^*(S) = q \oplus I \{ m_i(S_i) \}_I.S'_I$ implies $\forall k \in I: \exists \Gamma'', \Gamma''' \Gamma' \rightarrow^s \Gamma'' \text{ and } \Gamma'' \rightarrow^s \Gamma'''$, $[p]:S \rightarrow \Gamma''', [p]:S'_K$.
Note that in Def. K.7 is specifically based on [Bocchi et al. 2015, Def. 4]: clause 1 models a CFSM that is waiting from another CFSM in the system (i.e., a type in \( \Gamma' \)), and eventually succeeds; clause 2 models a CFSM that has queued a single output message, that is eventually received by another CFSM in the system. Also note the reductions in Def. K.7 mimic the "synchronous transition system" of the CFSMs in the original formulation: intuitively, it means that each message enqueued by a CFSM is immediately consumed by the recipient, and thus, at most one message can be queued at each reduction step, and must be received at the very next step. In our setting, such pairs of alternating queueing/dequeueing reductions are captured by a single synchronisation step.

**Proposition K.8.** Assume \( \text{dom}(\Gamma) = \{s\} \). Then, \( \Gamma \) is multiparty compatible if and only if \( \Gamma \) is live.

**Proof.** First, let \( \varphi = \text{beh}(\Gamma) \).

\((\Rightarrow)\) We show that when \( \Gamma \) is multiparty compatible, \( \varphi \) satisfies all clauses of Fig. 5(5), hence is a liveness property; then, by Prop.K.3, we conclude that \( \Gamma \) is live.

\((\Leftarrow)\) If \( \Gamma \) is live, \( \varphi \) is a liveness property (by Prop.K.3); therefore, by Def. K.1, \( \Gamma \rightarrow^* \Gamma' \) implies that \( \Gamma' \) is contained (with its unfoldings) in \( \varphi \), and satisfies the clauses of Fig. 5(5). By inspecting each \( \Gamma' \), we prove that it satisfies the clauses of Def. K.7; then, we conclude that \( \Gamma \) is multiparty compatible. \( \square \)

**Lemma K.9.** Assume \( \exists G : \text{fproj}_{G,q}(\Gamma) \). Then, \( \text{dom}(\Gamma) = \{s\} \) (Def. 2.6), and live(\( \Gamma \)).

**Proof.** We obtain \( \text{dom}(\Gamma) = \{s\} \) straightforwardly from Def. 5.8. The rest of the statement is consequence of [Deniélou and Yoshida 2013, Thm. 4.3]: if all projections of \( G \) onto its roles are defined (which is our hypothesis), then the resulting set of local types is multiparty compatible (Def. K.7); then, we conclude by Prop.K.8. \( \square \)

**Proposition K.10.** Assume that \( G|q \) is defined. Then, either:

(A) \( q \notin G \) and \( G|q \leq \text{end} \), or

(B) \( q \notin G \) and \( G|q = t \) (for some \( t \)), or

(C) \( \exists G, I, G_i (i \in I), p \) such that \( G = \bigoplus_i [G_i] \), \( q \notin G \), and either:

(a) \( \forall i \in I : G_i = p \rightarrow q: \{m_j(S_j) . G'_j\} \) \( j \in J_i \); or

(b) \( \forall i \in I : G_i = q \rightarrow p: \{m_j(S_j) . G'_j\} \) \( j \in J_i \).

**Proof.** By structural induction on \( G \):

- base case \( G = \text{end} \). Then, \( q \notin G \); moreover, we have \( (\mu t.G)|q = \text{end} \) (by Def. 3.3). Hence, we conclude by obtaining (A);

- base case \( G = t \) (for some \( t \)). Then, \( q \notin G \); moreover, we have \( G|q = t \) (by Def. 3.3). Hence, we conclude by obtaining (B);

- inductive case \( G = r \rightarrow r' : \{m_k(S_k) . G'_k\} \) \( k \in K \). Then, we have the following sub-cases:

  - \( r' = q \). Then, by letting \( \mathbb{G} = \{1\}, J = \{ \} \), \( G_1 = G \); we conclude by obtaining (C)(a);

  - \( r = q \). Similar to the previous case, and we conclude by obtaining (C)(b);

  - \( r \neq q \neq r' \). Then, we have:

\[
G|q = \bigcup_{k \in K} (G'_k)|q \tag{91}
\]

\( \forall k \in K : G'_k|q \) is defined \( \tag{92} \)

\( \forall k \in K : G_k \) satisfies either (A), (B) or (C), with \( G_k \) in place of \( G \) \( \tag{93} \)

Now, by inspecting the cases in which the merging in (91) is defined (by Def. 3.3), we can reduce (93) to the following possibilities:

* \( \exists \eta \geq 0 : \forall k \in K : G_k|q = \mu t_1 \cdots \mu t_n . \text{end} \). Then, \( G|q = \mu t_1 \cdots \mu t_n . \text{end} \) (by Def. 3.3), and thus, \( G|q \leq \text{end} \) (by Prop.1.15). Hence, we conclude by obtaining (A);

* \( \forall k \in K : G_k|q = t \) (for some \( t \)). Then, \( G|q = t \) (by Def. 3.3), and thus, we conclude by obtaining (B);

* \( \forall k \in K : \exists \mathbb{G}, I_k, G'_i (i \in I_k) \) such that \( G_k = \bigoplus_{i \in I_k} G'_i \), \( q \notin G_k \), and either:

  - \( \forall k \in K, i \in I_k : G'_i = p \rightarrow q: \{m_j(S_j) . G'_j\} \) \( j \in J_i \); Then, by letting \( \mathbb{G} = r \rightarrow r' : \{m_k(S_k) . G'_k\} \) \( k \in K \), \( I = \bigcup_{k \in K} I_k \), and \( \forall i \in I : G_i = G'_i \), we conclude by obtaining (C)(a);

  - \( \forall k \in K, i \in I_k : G_i = q \rightarrow p: \{m_j(S_j) . G'_j\} \) \( j \in J_i \). Similar to the previous case, and we conclude by obtaining (C)(b);
• inductive case $G = \mu t.G'$. Then, we have:

$$G|q = \begin{cases} \mu t.(G'|q) & \text{if } G'|q \neq t' \ (\forall t') \\ \text{end} & \text{otherwise} \end{cases}$$

(by Def. 3.3) \hspace{1cm} (94)

$G'$ is defined (by (94)) \hspace{1cm} (95)

$G'$ satisfies either (A), (B) or (C), with $G'$ in place of $G$ (by (95) and the induction hyp.) \hspace{1cm} (96)

Now, we have the following possibilities:

- $G'|q \leq \text{end}$. Then, $\exists n \geq 0 : G'|q = \mu t_1 \cdots \mu t_n.\text{end}$, which implies $G|q = \mu t.\mu t_1 \cdots \mu t_n.\text{end}$ (by Def. 3.3), and thus, $G'|q \leq \text{end}$ (by Prop. 1.15). Hence, we conclude by obtaining (A);

- $G'|q = t'$ (for some $t'$). Then, by (94) we have $G'|q = \text{and}$, and we conclude by obtaining (A).

- $\exists G', I', G'_i (i \in I')$ such that $G' = G'[G'_j]_{j \in I'}$, $q \notin \mathcal{G'}$, and either:
  * $\forall i \in I' : G'_i = p \rightarrow q : \{m_j(S_j) \cdot G'_j\}_{j \in I_i}$. Then, by letting $\mathcal{G} = \mu t.\mathcal{G'}$, $I = I'$, and $\forall i \in I : G_i = G'_i$, we conclude by obtaining (C)(a);  
  * $\forall i \in I' : G'_i = q \rightarrow p : \{m_j(S_j) \cdot G'_j\}_{j \in I_i}$. Similar to the previous case, and we conclude by obtaining (C)(b).

\[\Box\]

Corollary K.11. Let $G = \mu t.G'$, and assume that $G|q$ is defined. Then, either:

(A) $q \notin \mathcal{G}$ and $(\mu t.G)|q \leq \text{end}$, or
(B) $\exists \mathcal{G}, I, G_i (i \in I), p$ such that $G = \mu t.\mathcal{G}[G_i]_{i \in I}$, $q \notin \mathcal{G}$, and either:

(a) $\forall i \in I : G_i = p \rightarrow q : \{m_j(S_j) \cdot G'_j\}_{j \in I_i}$; or
(b) $\forall i \in I : G_i = q \rightarrow p : \{m_j(S_j) \cdot G'_j\}_{j \in I_i}$.

Proof. Assuming the hypothesis, we know that $G'|q$ is defined, and does not equal $t'$ for any $t'$ (by Def. 3.3). Therefore, we can apply Prop. K.10 on $G'$, obtaining either K.10(A), or K.10(C); from these, with the same reasoning of the case "$G = \mu t.G'" in the proof of Prop. K.10, we obtain respectively items (A) or (B) of the thesis. \[\Box\]

Proposition K.12. Assume that $T = G|q$ is defined. Then:

(1) $T = \mu t.T'$ implies $\exists \mathcal{G} : q \notin \mathcal{G}$ and $G = \mathcal{G}[\mu t.G_i]_{i \in I}$ and $\forall i \in I : G_i|q \neq t' \ (\forall t')$ and $T' = \bigcap_{i \in I} (G_i|q)$;

(2) $T = \mu t_1 \cdots \mu t_n. p \& \& j \in J \{m_j(S_j) \cdot S'\}_{j \in J}$ implies $\exists \mathcal{G} : q \notin \mathcal{G}$ and $G = \mathcal{G}[p \rightarrow q : \{m_j(S_j) \cdot G'_j\}_{j \in J}];$

(3) $T = \mu t_1 \cdots \mu t_n. p \& \& j \in J \{m_j(S_j) \cdot S'\}_{j \in J}$ implies $\exists \mathcal{G} : q \notin \mathcal{G}$ and $G = \mathcal{G}[q \rightarrow p : \{m_j(S_j) \cdot G'_j\}_{j \in J}];$

(4) $T = \mu t_1 \cdots \mu t_n. \text{end}$ or $T = t$ (for some $t$) implies $q \notin \mathcal{G}$.

Proof. Item (1) is proven by assuming

$$G|q = \mu t.T' \quad \text{(by hypothesis)}$$

and proceeding by structural induction on $G$:

• base cases $G = \text{end}$ and $G = t'$ (for some $t'$). Impossible, because by Def. 3.3, they would contradict (97);

• inductive case $G = \mu t.\mathcal{R} : \{m_j(S_j) \cdot G_j\}_{j \in J}$. By Def. 3.3, we have $p \neq q \neq r$ (otherwise, we would contradict (97)). Then:

$$G|q = \bigcap_{j \in J} (G_j|q) \quad \text{with } G_j|q = \mu t.T_j \quad \text{and } T' = \bigcap_{j \in J} T_j \quad \text{(by (97) and Def. 3.3)} \hspace{1cm} (98)$$

\[\forall j \in J : \ G_j|q = \mu t_j.T_j' \quad \text{implies } \ \exists \mathcal{G} : q \notin \mathcal{G}_{j} \quad \text{and } \ G_j = \mathcal{G}[\mu t_j.G'_j]_{j \in I_j} \quad \text{and } \ \forall i \in I_j : G'_i|q \neq t' \ (\forall t') \quad \text{and } \ T'_j = \bigcap_{i \in I_j} (G_i|q) \quad \text{(by i.h.)} \hspace{1cm} (99)\]

and letting $I = \bigcup_{j \in J} I_j$ and $\mathcal{G} = \mathcal{G} \rightarrow r : \{m_j(S_j) \cdot G_j\}_{j \in J}$, (by (99) and (98)) we get $G = \mathcal{G}[\mu t.G'_j]_{j \in I}$ and the rest of the thesis, and we conclude;

• inductive case $G = \mu t'G'$. Then, we have:

$$t' = t \quad \text{and } \ T' = G'|q \neq t'' \ (\forall t'') \quad \text{(by (97) and Def. 3.3)} \hspace{1cm} (100)$$

and letting $I = \{1\}$, $G_1 = G'$ and $\mathcal{G} = [\ ]^1$, (by (100)) we get $G = \mathcal{G}[\mu t.G_i]_{i \in I}$ and the rest of the thesis, and we conclude.
For the other items, we have:

\[ G | q \text{ is defined} \quad \text{(by hypothesis)} \quad \text{(101)} \]

\[ \text{either} \quad q \notin G \quad \text{(by (101) and Prop. K.10)} \quad \text{(102)} \]

or

\[ G = \bigcirc[G_i]_{i \in I'} \quad \text{with} \quad q \notin \bigcirc \quad \text{and for some} \; p, \text{either:} \]

\[ \forall i \in I': G_i = p \rightarrow q: \{ m_j(S_j) \cdot G'_j \} \quad \text{(by (101) and Prop. K.10)} \quad \text{(103)} \]

and

\[ G_i[q = p \& j \in j_i m_j(S_j)] \cdot S'_j \quad \text{where} \; S'_j = G'_j[q] \quad \text{(by (104) and Def. 3.3)} \quad \text{(104)} \]

or

\[ \forall i \in I': G_i = q \rightarrow p: \{ m_j(S_j) \cdot G'_j \} \quad \text{(by (105) and Def. 3.3)} \quad \text{(105)} \]

Note that (101) rules out cases \((A)\) and \((B)\) of Prop. K.10, and thus leaves us with either (104) or (105).

Using (101) and (103), we rephrase the statement with \( \bigcirc[G_i]_{i \in I'} \) in place of \( G \), and proceed by structural induction on \( G \). The results follow by item (1) and Def. 3.3.

**Proposition K.13.** Assume \( \text{dom}(\Gamma) = \{ s \} \), and \( \Gamma \rightarrow \Gamma_1 \rightarrow \Gamma_2 \rightarrow \cdots \rightarrow \Gamma_n = \Gamma \) (with \( n \geq 1 \)), and:

\[ \forall \Gamma', \Gamma'' : \left\{ \Gamma = \Gamma', \Gamma'' \text{ and } \exists \Gamma_1', \ldots, \Gamma_n' : \Gamma' \rightarrow \Gamma_1' \rightarrow \cdots \rightarrow \Gamma_n' = \Gamma'' \right\} \text{ implies } \Gamma'' = \emptyset \]

Then, \( \forall s[p]: S \in \Gamma : S \not\in \text{end} \) and \( S \) contains a recursive sub-term \( \mu t.S' \) (for some \( t, S' \)).

**Proof.** By contradiction, assume that:

\[ \exists s[p]: S \in \Gamma \quad \text{such that} \quad S \not\leq \text{end} \quad \text{or} \quad S \text{ has no recursive sub-term} \quad \text{(106)} \]

Then, we have two possibilities:

\( a \) if the entry \( s[p]: S \) interacts with other entries at least once along \( \Gamma \rightarrow \Gamma_1 \rightarrow \cdots \rightarrow \Gamma \), then it cannot reduce into \( s[p]: S \) (by (106) and Def. 2.8); therefore, \( \Gamma \) cannot not reduce into itself, thus contradicting the hypothesis \( \Gamma \rightarrow \Gamma_1 \rightarrow \cdots \rightarrow \Gamma \);

\( b \) otherwise, if the entry \( s[p]: S \) does not interact with other entries along \( \Gamma \rightarrow \Gamma_1 \rightarrow \cdots \rightarrow \Gamma \), then we can find \( \Gamma'' \) (as defined in the statement) such that \( s[p]: S \in \Gamma'' \), thus contradicting the hypothesis \( \Gamma'' = \emptyset \).

Therefore, assuming (106) leads to a contradiction; hence, we conclude that \( \forall s[p]: S \in \Gamma : S \not\in \text{end} \) and \( S = \mu t.S' \) (for some \( t, S' \)).

**Proposition K.14.** Assume that \( S \) has a recursive subterm \( \mu t.S' \), for some \( t \) and \( S' \). Then:

\( 1 \) if \( t \in \text{fv}(S') \), then \( \mu t.S' \) is a subterm of \( \text{unf}^n(S) \) (for all \( n \));

\( 2 \) \( \forall \Gamma_0, \Gamma, \Gamma, s, p : \text{if} \; \Gamma_0, s[p]: S_0 \rightarrow \Gamma, s[p]: S \; \text{then} \; \mu t.S' \) is a subterm of \( S_0 \).

Moreover, for all \( S \), if \( \mu t.S' \) is a subterm of \( \text{unf}^n(S) \) (for some \( n \)), then \( \mu t.S' \) is also a subterm of \( S \).

**Proof.** Item (1): by structural induction on \( S \), and then by induction on \( n \).

Item (2): by induction on the number of reductions, showing that \( \mu t.S' \) is a subterm of the type of \( s[p] \), in each predecessor of \( \Gamma \), \( s[p]: S \).

The "moreover..." part of the statement is proven by first showing that

\[ \forall S_0 : \text{if} \; \mu t.S' \text{ is a subterm of } \text{unf}^1(S_0), \text{then it is also a subterm of } S_0 \quad \text{(107)} \]

and then proceeding by induction on \( n \), using (107) in the inductive case.

**Theorem K.15.** Assume \( \exists G, \Gamma : f\text{proj}_{G, s}(\Gamma) \). Then, \( \text{dom}(\Gamma) = \{ s \} \) (Def. 2.6), and \( \text{live}^+ (\Gamma) \).

**Proof.** Assuming the hypotheses, we have (for some \( G \)):

\[ \Gamma = \{ s[p] : G[p] \} \quad \text{peroles}(G) \quad \text{(by Def. 5.8)} \quad \text{(108)} \]

\[ \text{dom}(\Gamma) = \{ s \} \quad \text{(by (108))} \quad \text{(109)} \]

\[ \text{live}(\Gamma) \quad \text{(by Lemma K.9)} \quad \text{(110)} \]

Note that (109) proves the first part of the thesis. To prove the second part, let:

\[ \varphi = \text{beh}(\Gamma) \quad \text{(111)} \]

We now show that \( \varphi \) is a liveness* property. Therefore, we examine each element \( (\Gamma', s[p]: S) \in \varphi \), and we show that it satisfies all clauses of Fig. 5(6). We have the following (non-mutually exclusive) possibilities:
In this case, clauses \([L->]^*\) and \([L->^*\) of Fig. 5(6) are vacuously satisfied, and we are left to prove clause \([L->^*\). The first part of such a clause (i.e., the part that matches clause \([L-\&\) of Fig. 5(5)) holds by (110), (111) and Prop. K.3.

We now prove the "moreover..." part of clause \([L-\&\). We need to prove that:

\[ Γ', s[π]:S \text{ belongs to some fair traversal set } X \text{ (Def. 5.5) with targets } Y \text{ such that, } ∀Γ_τ ∈ Y, \text{ we have } Γ_τ = Γ'', s[π]:S_k' \]  \hspace{1cm} (112)

We proceed by contradiction, assuming that:

\[ \text{there is no fair traversal set that contains } Γ', s[π]:S, \text{ and has targets } Y \text{ such that, } ∀Γ_τ ∈ Y, \text{ we have } Γ_τ = Γ'', s[π]:S_k' \]  \hspace{1cm} (113)

This means that there is \(Γ_0\) such that:

\[ Γ_0 ⊆ Γ' \]  \hspace{1cm} (114)
\[ ∃Γ'_0, Γ''_0, k ∈ I : Γ_0 → Γ'_0 \text{ and } Γ'_0, s[π]:S → Γ''_0, s[π]:S_k' \]  \hspace{1cm} (115)
\[ ∃n ≥ 1, Γ_{00}, Γ_{01},..., Γ_{0n} : Γ_0 → Γ_{00} → Γ_{01} → ... → Γ_{0n} → Γ_0 \]  \hspace{1cm} (116)

i.e., \(Γ_0\) is a subset of \(Γ'\) (114) that can interact with \(s[π]:S\) (115), but can also loop indefinitely without interacting with \(s[π]:S\) (116); we also pick \(Γ_0\) to be minimal, in the sense that if we remove any entry, then (115) does not hold. The combination of minimality, (115) and (116) is due to liveness (110) and (113), that claims the impossibility to construct a fair traversal set with targets that always interact with \(s[π]:S\) (similarly to Ex. 5.6).

Now, take \(Γ_{00}\) in (116), and partition it as \(Γ_{00} = Γ_1, Γ_2\) such that:

\[ ∃Γ_{10}, Γ_{11},..., Γ_{1n} : Γ_1 = Γ_{10} → Γ_{11} → Γ_{12} → ... → Γ_{1n} → Γ_1 \text{ and } ∀i ∈ 0..n : Γ_{0i} = Γ_{1i}, Γ_{2i} \]  \hspace{1cm} (117)

\[ ∀Γ'_1, Γ''_1 : (Γ_1 = Γ'_1, Γ''_1 \text{ and } (∃Γ'_{10},...,Γ'_{1n} : Γ'_1 = Γ'_{10} → Γ'_{11} → ... → Γ'_{1n} → Γ'_1)) \text{ implies } Γ''_1 = \emptyset \]  \hspace{1cm} (118)

i.e., we pick \(Γ_1, Γ_2\), so that \(Γ_1\) induces the reductions in (116), and reduces into itself, without interacting with \(Γ_2\) (by (117)); moreover, we are picking \(Γ_1\) so that it is minimal w.r.t. the reductions in (117), i.e., all its entries reduce at least once (by (118)). This implies that:

all entries of \(Γ_1\) contain a recursive subterm \hspace{1cm} (by (117), (109), (118) and Prop. K.13) \hspace{1cm} (119)

Now, by (115) and (117), we have two mutually exclusive sub-cases:

(A) \(∃q' : s[π]:S'_q \subseteq Γ_1\). Then, we have:

\[ S'_q \text{ has a subterm } S'_q'' = μT. S[T_j]_{j ∈ J} \] \hspace{1cm} (by (119)) \hspace{1cm} (120)

\[ S''_q \text{ is a subterm of } S''_q \text{ with } T_k = p ⊕ i ∈ I_k m_i(S_i).S'_i \] \hspace{1cm} (by (115), (117) and Def. 2.8) \hspace{1cm} (121)

\[ S' \text{ is a subterm of } (μT. S'[T_j]_{j ∈ J}) \] \hspace{1cm} (by (120), (121) and (122)) \hspace{1cm} (123)

Now, observe:

\[ S''_q = S'' \text{ is a subterm of } S \text{ and } S = Γ(s[π]) = G \text{ with } S = Γ(s[π]) = G \] \hspace{1cm} (by (108)) \hspace{1cm} (124)

\[ S''' \text{ is a subterm of } G \text{ and } S''' = (μT.G') \text{ with } \muT.G' \] \hspace{1cm} (by (125), Prop. K.12(1) and Def. 3.3) \hspace{1cm} (125)

\[ p ∈ S' \text{ and therefore } p ∈ μT.G' \] \hspace{1cm} (by (120), (121), (126) and Def. 3.3) \hspace{1cm} (127)

and note that in \(G'\) there must be:

(A-1) a branch where the first interaction of \(p\) is either \(p → q\) or \(q → p\) since, by hypothesis, \(s[π]:S\) is waiting for \(q\), and by Prop. K.12); and
(A-2) a branch where \( p \) does not interact with \( q \) (otherwise, by Def. 3.3, we could not project \( S'_q \) according to (126) and (123)).

But then:

\[
\begin{align*}
\mathcal{G}, q', I, G_i (i \in I) : \quad & \left\{ \begin{array}{ll}
 ministers \varepsilon_i G' = \varepsilon_i G_i \mid I_i \\
 p \notin \varepsilon_i G
\end{array} \right.
\end{align*}
\]

and either

\[
\forall i \in I : \quad G_i = p \rightarrow q'; \left\{ m_j(S_j) \cdot G'_j \right\} j \in I_i
\]

or

\[
\forall i \in I : \quad G_i = q' \rightarrow p'; \left\{ m_j(S_j) \cdot G'_j \right\} j \in I_i
\]

which implies:

\[
\begin{align*}
(p \cdot G') | p & \text{ is undefined} & \text{(by (128), (127), and the contrapositive of Cor.K.11)} \\
G | p & \text{ is undefined} & \text{(by (129), (126) and Def. 3.3)}
\end{align*}
\]

and (130) contradicts (124), and thus, the hypothesis (108);

(B) \( s[q] \in \text{dom}(G_2) \). Then, we have two more sub-cases:

(a) \( G_2 \) can first interact with \( G_1 \), and then with \( s[p]: S \). More formally:

\[
\exists k \in I, I_1', I_2' : \quad G_1, G_2 \rightarrow \rightarrow s. I_1', I_2' \quad \text{and} \quad G'_2, s[p]: S \rightarrow G'_2, s[p]: S'_k
\]

where in the first sequence of reductions, an entry of \( G_2 \) (say, \( s[r'] \cdot S_r' \)) interacts with an entry of \( G_1 \) (say, \( s[r] \cdot S_r \)). But then, we can apply the same reasoning of case (A) above, with the following adaptations:

(i) use \( r' \) in place of \( q' \);

(ii) use \( r \) in place of \( p \);

(iii) if \( S_r' \) is a (possibly recursive) external choice, let \( T_k \) in (121) be an external choice, too.

Then, the adaptation of (130) says that \( G | r \) is undefined, contradicting the hypothesis (108);

(b) \( G_2 \) can first interact with \( s[p]: S \), and then with \( G_1 \). More formally:

\[
\exists k \in I, I_1', I_2', I_3', I_4', S', r, S_r, S'_r, S''_r : \quad s[r] \in \text{dom}(G_2) \quad \text{and} \quad G_2, s[p]: S \rightarrow \rightarrow s. I_2', S_r, s[p]: S_r \rightarrow \rightarrow s. I_1', I_2', I_3', I_4', S'_r, S''_r
\]

where in the last sequence of reductions, \( s[r]: S_r' \) interacts with some element of \( G_1 \) — say, \( s[r'] : S_r' \). But then, we can use the same strategy described in case (a) above: i.e., apply the reasoning of case (A) with the adaptations (i), (ii) and (iii), obtaining the same contradiction;

To recap: in all cases above, assuming (113) leads to a contradiction, hence we obtain that its negation (112) holds. Therefore, we conclude that \( \Gamma' \), \( s[p]: S \) satisfies clause \([L \& \cdot] \) of Fig.5(6):

- \( S = q \otimes \otimes I_i \in m_i(S_1) \cdot S'_1 \). In this case, clauses \([L \& \cdot \cdot] \) and \([L \& \cdot \cdot \cdot] \) of Fig.5(6) are vacuously satisfied, and we are left to prove clause \([L \& \cdot \cdot \cdot \cdot] \). Its proof is similar to the previous case.

- \( S = \mu s'. \). In this case, clauses \([L \& \cdot \cdot \cdot \cdot] \) and \([L \& \cdot \cdot \cdot \cdot \cdot] \) of Fig.5(6) are vacuously satisfied, and we are left to prove clause \([L \& \cdot \cdot \cdot \cdot \cdot \cdot] \) — which holds by Equation (111) and Prop.K.2;

- \( S = \text{end} \). In this case, clauses \([L \& \cdot \cdot \cdot \cdot \cdot \cdot] \) and \([L \& \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot] \) hold vacuously;

- \( \Gamma', s[p]: S \rightarrow \rightarrow \Gamma'' \). In this case, we also need to satisfy clause \([L \rightarrow \rightarrow \cdot] \) of Fig.5(6) — which holds by Equation (111) and Prop.K.2.

Summing up, we have proven that if we take any \( G \) whose projections are defined, we can also define \( \varphi \) as in (111), and prove that it is a liveness\(^+ \) property. Moreover, \( \{s[p]: G | p | \}_{p \in \text{Eores}(G)} \in \varphi \) (by (108), (111) and Def. K.1); and since \( \text{live}^+ \) is the largest liveness\(^+ \) property (by Fig.5(6)), we have \( \varphi \subseteq \text{live}^+ \), and thus, \( \{s[p]: G | p | \}_{p \in \text{Eores}(G)} \in \text{live}^+ \). Therefore, by Def. 5.8 we conclude that if \( \exists \mathcal{G}, \Gamma : fpro_{G, \alpha}(\Gamma) \), then \( \text{live}^+(\Gamma) \).

\[\square\]

**Lemma 5.9.** For all \( \Gamma \), the following (non-)implications hold:
(1) consistent(Γ) ⇔ safe(Γ);
(2) live(Γ) ⇔ safe(Γ);
(3) live(Γ) ⇔ df(Γ);
(4) nterm(Γ) ⇔ df(Γ);
(5) consistent(Γ) ⇔ df(Γ);
(6) consistent(Γ) ∧ df(Γ) ⇔ live(Γ);
(7) live++(Γ) ⇔ live*(Γ) ⇔ live(Γ);
(8) term(Γ) ⇔ live++(Γ);
(9) assume dom(Γ) = {s} (Def. 2.6). Then:
∃G : fproj(s,Γ) ⇔ live*(Γ).

**Proof.** The negated implications in the statement are proved in Table 1 and Ex.5.11.
We now examine the remaining implications.

1. Assume Γ consistent, and take the property φ = beh(Γ) (Def. K.1). By contradiction, assume that Γ is not safe. Then, by Prop. K.3, φ must contain some Γ′ such that Γ →* Γ′ and Γ′ violates clause [S-φ] (possibly after applying [S-µ] to unfold its entries). By Def. 3.8, such Γ′ is not consistent. But then, by the contrapositive of Prop. K.5, we obtain that Γ is not consistent — contradiction. We conclude that Γ is safe.

2. Straightforward by Fig.5(5).
3. Straightforward by Fig.5(5).
4. Straightforward by Fig.5(4).
5. Straightforward by Fig.5(6) and Fig.5(7).
6. By contradiction, assume that Γ is not live++. Then, there is a Γ′ such that Γ →* Γ′, and Γ′ (once unfolded) violates clause [L-φ] of Fig.5(7) — i.e., Γ′ = Γ″, s[p];S (for some Γ″), where S is a branching/selection type that is not triggered within a finite number of steps by a corresponding selection/branching along the reductions of Γ″. But then, there is no guarantee that, in a finite number of steps, Γ′ will reduce to some Γ‴ such that end(Γ‴); and since Γ →* Γ′, there is no such guarantee for Γ, either. This implies that Γ does not satisfy Fig.5(3) — contradiction. Therefore, we conclude that Γ is live++.


**Theorem 5.13 (Decidability of φ).** φ(Γ) is decidable, for all Γ, and for all φ such that
φ ∈ \{consistent, fproj_{G,s}, pproj_{G,s}, safe, term, nterm, df, live, live*, live++, live++\} (for any G)

**Proof.** If φ = consistent, Γ ∈ φ is decidable because, by Def. 3.8, it is sufficient to check (at most) all pairs of types contained in Γ (which are finite), using partial projection (that always terminates), duality and ≤ (that are both decidable).

For the other cases, observe that for any Γ, the transitive closure of the typing context reduction relation → (Def. 2.8) induces a finite-state transition system.

When φ = fproj_{G,s} or φ = pproj_{G,s}, observe the two possible definitions of the set E in Def. 5.8: in both cases, the projection of any G always terminates, either by being undefined (then E is empty, φ is empty, and φ(Γ) is false) or by returning some Γ, from which Def. K.1 collects all reachable typing contexts, that are finite. Then, notice that unf* (E) in Def. K.1 is the least fixed point of a function that is monotonic (w.r.t. the partial order ≤), and therefore, can be computed with a chain of successive applications starting with Ø (by the Knaster-Tarski theorem [Tarski 1955]) — and since E is finite, this procedure always terminates by yielding the (finite) set of typing contexts with all combinations of all unfoldings of all elements of all typing contexts contained in E. Hence, φ is finite, and therefore, Γ ∈ φ is trivially decidable.

The rest of the cases are decidable because it is straightforward to produce an algorithm that inspects all (finite) elements of the behavioural set beh(Γ) (Def. K.1), verifying whether they satisfy the clauses of Def. 4.1, Fig.5(2), Fig.5(5) or Fig.5(6).

## L ASYNCHRONOUS SESSION FIDELITY

**Proposition L.1 (Asynchronous Normal Form).** For all P, P ≡ def D in (νβ) P_1 | ... | P_N, where ∀i ∈ 1..n, P_i is either a branching, a selection, or a process call, or a session queue.
Proof. Minor adaptation of Prop.1.2. □

Lemma B.1 (substitution) is unchanged in async MPST, it is only applied to processes occurring as premises of [TA-Lrr]. Instead, Lemma B.2 (subject congruence) needs to be adapted as Lemma L.2 below. The difference w.r.t. the synchronous result is that, in the asynchronous setting, if $P \equiv P'$ then session queues might be reordered by $\equiv$ (Fig.7), hence queue types might need reordering by $\equiv$ (Def. D.2).

**Lemma L.2 (Async Subject Congruence).** Assume $\Theta \cdot \Gamma \vdash s$ and $P \equiv P'$. Then, $\exists \Gamma'$ such that $\Gamma \equiv \Gamma'$ and $\Theta \cdot \Gamma' \vdash s$. Proof. The proof is similar to that of Lemma B.2, and in all corresponding cases we have $\Gamma = \Gamma'$. We have two additional cases, corresponding to the two queue congruence rules in Def. C.1:

- when $P = (\forall s: \Gamma''')s \triangleright \sigma \equiv 0 = P'$, we must have end($\Gamma'''$) and end($\Gamma$), and we conclude with $\Gamma' = \Gamma$ by typing rule [T-G]
- when $P \equiv P'$ holds by the order-swapping congruence on queues, we apply the same reordering on the queue types of $\Gamma$, getting a congruent typing context $\Gamma'$ that satisfies the statement. □

**Proposition L.3.** If $p \in \text{senders}(s)$, then $\forall \Theta, \Gamma, s : \Theta \cdot \Gamma \vdash s \triangleright \sigma$ implies $\Gamma(s[p]) = M$, for some $M \neq \epsilon$.

Proof. Assume $p \in \text{senders}(s).$ Then:

\[
\exists \sigma', \sigma'' : \sigma = \sigma' \cdot (p, q, m(s'[r])) \cdot \sigma'' \quad \text{(by hypothesis and Def. C.1)} \quad (133)
\]

\[
\Theta \cdot \Gamma \vdash s \triangleright \sigma'' \quad \Gamma' \vdash s'[r] : S \\
\Theta \cdot (\Gamma \triangleright s[p] : q \cdot \text{Im}(S) \cdot \epsilon) \vdash s \triangleright (p, q, m(s[r])) \cdot \sigma'' \quad \text{[TA-\sigma]} \quad (134)
\]

From (134), we proceed with a further induction on $\sigma'$ (from (133)), to prove that $\exists \Gamma$ such that $\Theta \cdot \Gamma \vdash s \triangleright \sigma$, with $\Gamma(s[p]) = M$ for some $M \neq \epsilon$:

- base case $\sigma' = \epsilon$. Then, $\sigma = \sigma' \cdot (p, q, m(s'[r])) \cdot \sigma'' = (p, q, m(s'[r])) \cdot \sigma''$: we conclude by (134) and Fig.8, letting $\Gamma = (\Gamma \triangleright s[p] : q \cdot \text{Im}(S) \cdot \epsilon, \Gamma')$;
- inductive case $\sigma' = (p', q', m'(s'[r'])) \cdot \sigma'''$. Then:

\[
\exists \sigma' : \Theta \cdot \Gamma' \vdash s \triangleright \sigma'' \cdot (p, q, m(s'[r])) \cdot \sigma'' \quad \text{with} \quad \Gamma''(s[p]) = M' \quad \text{(by i.h.)} \quad (135)
\]

\[
\Theta \cdot \Gamma'' \vdash s[p] : q' \cdot \text{Im}'(S'') \cdot \epsilon, \Gamma'' \vdash s \triangleright (p', q', m'(s'[r'])) \cdot \sigma'' \quad \text{[TA-\sigma]} \quad (136)
\]

and we conclude by (136) and Fig.8, letting $\Gamma = (\Gamma'' \triangleright s[p'] : q' \cdot \text{Im}'(S'') \cdot \epsilon, \Gamma'')$. □

**Proposition L.4.** If $p \notin \text{senders}(s)$, then $\forall \Theta, \Gamma, s : \Theta \cdot \Gamma \vdash s \triangleright \sigma$ implies $s[p] \notin \text{dom}(\Gamma)$.

Proof. Assume $p \notin \text{senders}(s).$ We proceed by induction on $\sigma$:

- base case $\sigma = \epsilon$. Then, we have $\Gamma = \emptyset$ (by inversion of [TA-\epsilon]), and we conclude immediately;
- inductive case $\sigma = (p', q, m(s[r])) \cdot \sigma'$. Observe that $p' \neq p$ and $p \notin \text{senders}(\sigma')$ (otherwise, we would have the contradiction $p \in \text{senders}(\sigma)$). By the i.h., $\forall \Theta, \Gamma', s : \Theta \cdot \Gamma' \vdash s \triangleright \sigma'$ implies $s[p] \notin \text{dom}(\Gamma')$. By [TA-\sigma] and Fig.8, we conclude that $\forall \Theta, \Gamma, s : \Theta \cdot \Gamma \vdash s \triangleright \sigma$ implies $s[p] \notin \text{dom}(\Gamma)$. □

Using Lemma B.1 (plus Lemma L.2) we extend Thm.B.4 (session inversion) to asynchrony, obtaining the new Thm. L.5 below: the difference w.r.t. Thm. B.4 are the new items (5)–(7), showing how queue types shape process queues.

**Theorem L.5 (Async Session Inversion).** Assume $\Theta \cdot \Gamma \vdash s \mid p \in P_p$ with each $P_p$ either being $\Theta$ (up-to $\equiv$), or only playing role $p$ in $s$. Then, $\Gamma = \{ s[p] : S_p \mid p \in P_p \} \cup \{ s[p] : M_p \mid p \in P_p \}$ for some $I', I''$. Moreover, $\forall p \in I'$, we have items (1), (2), (3) from Thm. B.4.

Further, we have item (4) from Thm. B.4, and (5) $\forall p \in I \setminus I'' : p \neq \text{senders}(\sigma)$.

Finally, $\forall p \in I'' : (6) \exists q, m, T, M' : M_p = q \cdot \text{Im}(T) \cdot M', (7) \exists s', r, \sigma' : \sigma \equiv (p, q, m(s'[r])) \cdot \sigma'$. Proof. The proof is similar to that of Thm. B.4, using the asynchronous MPST normal form of Prop. L.1, and observing that, by inversion of [TA-\sigma], we must have $S = \{ s \}$. We examine the additional items, involving the session queue.
Item (5). Holds by the contrapositive of Prop. L.3.
Item (6). By the contrapositive of Prop. L.4, we know that \( p \in \text{senders}(\sigma) \); then, we conclude by Def. C.1; Item (7). By the contrapositive of Prop. L.4, we know that \( p \in \text{senders}(\sigma) \); then, we conclude by Def. C.1.

\[ \square \]

Theorem F.8 (Async Session Fidelity). Let \( \Theta \cdot \Gamma \vdash_S P \), with \( P \equiv \left[ \bigcup_{p \in \Gamma} (s \triangleright \sigma) \mid s \triangleright \sigma \right] \), and each \( P_p \) either being 0 (up-to \( \equiv \)), or only playing role \( p \) in \( s \). Then, \( \Gamma \rightarrow_S \) implies \( \exists \Gamma', P' \) such that \( \Gamma \rightarrow_S \Gamma' \), \( P \rightarrow_S P' \) and \( \Theta \cdot \Gamma' \vdash_S P' \), with \( P' \equiv \left[ \bigcup_{p \in \Gamma'} (s \triangleright \sigma') \mid s \triangleright \sigma' \right] \), and each \( P'_p \) either being 0 (up-to \( \equiv \)), or only playing role \( p \) in \( s \).

Proof. Similar to the proof of Thm. 5.4, but using Thm. 4.5 for asynchronous session inversion.

\[ \square \]

M SUBJECT REDUCTION FOR ASYNCHRONOUS MPST

Remark M.1. If we want to explicitly instantiate the safety property \( \varphi \) for a typing derivation that restricts the multiparty sessions \( s_1: \Gamma_1, \ldots, s_n: \Gamma_n \), then we can (1) take a set \( \{ \varphi_i \}_{i \in [1..n]} \) where \( \varphi_i \) is an \( \{ s_i \} \)-safety property such that \( \varphi_i(\Gamma_i) \), and (2) instantiate Def. F.4 with \( \varphi = \bigcup_{i \in [1..n]} \varphi_i \). By construction, \( \varphi \) is an \( \{ s_i \} \)-safety property such that \( \varphi(\Gamma_i) \) (\( i \in [1..n] \)).

Lemma M.2. If \( a\text{-safe}_S(\Gamma) \) and \( \Gamma \rhd \Gamma' \), then \( a\text{-safe}_S(\Gamma') \).

Proof. Similar to Lemma 4.5, noticing that by Def. 4.1, subtyping of asynchronous typing contexts only involves session types (not queue types).

\[ \square \]

Lemma M.3. If \( \Gamma \) safe and \( \Gamma \leq S \Gamma'' \), then there is \( \Gamma''' \) such that \( \Gamma \rightarrow_S \Gamma'' \rightarrow_S \Gamma''' \).

Proof. Similar to Lemma 4.4.

\[ \square \]

Lemma M.4 (Narrowing (asynchronous)). If \( \Theta \cdot \Gamma \vdash_S P \) and \( \Gamma' \leq S \Gamma \) with \( a\text{-safe}_S(\Gamma') \), then \( \Theta \cdot \Gamma' \vdash_S P \).

Proof. Similar to Lemma B.3, noticing that by Def. 4.1, subtyping of asynchronous typing contexts only involves session types (not queue types).

\[ \square \]

Proposition M.5. Assume \( \Theta \cdot \Gamma \vdash_S \Gamma \vdash_S s_\Gamma \vdash_S \Gamma' \vdash_S s_\Gamma' \vdash_S \Gamma'' \vdash_S s_\Gamma'' \vdash_S s_\Gamma''' \vdash_S \Gamma''' \vdash_S s_\Gamma''\). Then, we have the judgement \( \Theta \cdot \Gamma, s[p] : q[m(S)] : \varepsilon, \Gamma' \vdash_S (s[p], q, m(s'[r])) \).

Proof. Assume \( \Theta \cdot \Gamma \vdash_S s_\Gamma \vdash_S \Gamma' \vdash_S (s[p], q, m(s'[r])) \). Notice that:

\[
\begin{align*}
\Theta \cdot \Gamma \vdash_S s_\Gamma & \vdash_S \varepsilon \tag{T-\varepsilon} \\
\Theta \cdot \Gamma \vdash_S s[p] : q[m(S)] : \varepsilon, \Gamma' \vdash_S (s[p], q, m(s'[r])) & \vdash_S (s[p], q, m(s'[r])) : \varepsilon \tag{T-\varepsilon}
\end{align*}
\]

Now, let \( n \) be the length of \( \sigma \), and let:

\[
\begin{align*}
\Gamma_0 & = \emptyset \\
\sigma_0 & = \varepsilon \\
\Gamma_n & = \Gamma \\
\sigma_n & = \sigma \\
\forall i \in [1..n] & : \sigma_i = (p_i, q_i, m_i(s_i[r_i])) \cdot \sigma_{i-1} \quad \text{and} \quad \Gamma'_i \vdash_s s_i[r_i] \vdash_S \gamma_i \quad \text{and} \quad \Gamma_i = (\Gamma_{i-1} \leftrightarrow_S s[p_i] : q_i[m_i(s_i[r_i])] : \varepsilon, \Gamma_i').
\end{align*}
\]

Then, the derivation of \( \Theta \cdot \Gamma \vdash_S s_\Gamma \vdash_S \Gamma' \vdash_S (s[p], q, m(s'[r])) \) has the following shape:

\[
\begin{align*}
\Theta \cdot \Gamma_0 \vdash_S s_\Gamma & \vdash_S \sigma_0 \tag{T-\sigma_0} \\
\Theta \cdot \Gamma_1 \vdash_S s_\Gamma & \vdash_S s_1[r_1] \vdash_S \sigma_1 \tag{T-\sigma_1} \\
\vdots \\
\Theta \cdot \Gamma_n \vdash_S s_\Gamma & \vdash_S s_n[r_n] \vdash_S \sigma_n \tag{T-\sigma_n} \\
\Theta \cdot \Gamma_n \vdash_S s_\Gamma & \vdash_S (s[p], q, m(s'[r])) \tag{T-\varepsilon}
\end{align*}
\]

We can rewrite the derivation in (138) into a derivation for \( \Theta \cdot \Gamma, s[p] : q[m(S)] : \varepsilon, \Gamma' \vdash_S (s[p], q, m(s'[r])) \), proceeding by induction on \( n \):

1. first, \( \forall i \in [0..n] \), rewrite \( \Theta \cdot \Gamma_i \vdash_S s_\Gamma \vdash_S \sigma_i \) as \( \Theta \cdot \Gamma_i, s[p] : q[m(S)] : \varepsilon, \Gamma' \vdash_S s_\Gamma \vdash_S \sigma_i \cdot (s[p], q, m(s'[r])) : \varepsilon. \) Notice that the rewritten typing context is defined, since \( \text{dom}(\Gamma_i) = \{ s_i[r_i] \} \nsubseteq \text{dom}(\Gamma_i) \) and \( s[p] \notin \text{dom}(\Gamma_i) \) (by hypothesis);
(2) then, graft (137) on top of the derivation, noticing that the conclusion of (137) matches the rewriting of $\Theta \cdot \Gamma_0 \vdash_s s \triangleright \sigma_0$ after step 1 above.

We have thus obtained a typing derivation that proves the statement. □

**Proposition M.6.** Assume $\Theta \cdot \Gamma \vdash_{\{s\}} s \triangleright \sigma$, with $s'[r] \notin \text{dom}(\Gamma)$, $\Gamma(s[p]) = M$ (for some $M$) and $\Gamma' \vdash s'[r] : S$. Then, we have $\Theta \cdot \Gamma \{ M \cdot q[m(S) \cdot \epsilon] / s[p] \}, \Gamma' \vdash_{\{s\}} s \triangleright \sigma \cdot (p, q, m(s'[r])).$

**Proof.** The proof is similar to that of Prop.M.5. The only difference is that, in step 1 of the rewriting, the observation “$\forall i \in 0..n : s[p] \notin \text{dom}(\Gamma_i)$” does not hold. Hence, we use the following rewriting, for all $i \in 1..n$:

$$\Theta \cdot \Gamma \vdash_{\{s\}} s \triangleright \sigma_i \quad \leftrightarrow \quad \Theta \cdot (s[p] \cdot q[m(S) \cdot \epsilon] \rightarrow \Gamma_i), \Gamma' \vdash_{\{s\}} s \triangleright \sigma_i \cdot (p, q, m(s[r])) \cdot \epsilon$$

where:

$$s[p] : M'' \rightarrow \Gamma''$$

as a result, in the rewritten derivation, the type $q[m(S)]$ for the additional queue message $(p, q, m(s'[r]))$ is added at the end of $\Gamma(s[p])$ from the original typing context. □

**Lemma M.7 (Queueing/Dequeueing Typability).** Assume $\Theta \cdot \Gamma \vdash_S P$ with $\Gamma$ $S$-safe. Then:

1. if $P = s[p][q] \oplus m(s'[q'])$. $P' \mid s \triangleright \sigma$, then $\exists \Gamma''$ $S$-safe such that $\Gamma \rightarrow_S \Gamma''$ and $\Theta \cdot \Gamma'' \vdash_S P'$ $\mid s \triangleright \sigma \cdot (p, q, m(s'[q']))$;
2. if $P = s[q][p] \sum_{i \in \mathcal{M}(x_i)}. P'_i \mid s \triangleright (p, q, m(s'[q'])). \sigma$ then there exist $k \in I$ and $\Gamma''$ $S$-safe such that $\Gamma \rightarrow_S \Gamma''$ and $\Theta \cdot \Gamma'' \vdash_S P'_k \{ s[q] / x_k \} \mid s \triangleright \sigma$.

**Proof.** **Item 1.** We have:

$$\Gamma = \Gamma_{\oplus}, \Gamma_{\sigma}, \text{ and } \Theta \cdot \Gamma_{\oplus} \vdash_{S_{\oplus}} s[p] \cdot q \oplus m(s'[q']). P' \mid s \triangleright \sigma \text{ if } S_{\oplus} \cap S_{\sigma} = \emptyset$$

$$\Theta \cdot \Gamma_{\oplus} \vdash_S s[p][q] \oplus m(s'[q']). P' \mid s \triangleright \sigma \text{ if } \Theta \cdot \Gamma_{\oplus} \vdash_{S_{\oplus}} s[p] \cdot q \oplus m(s'[q']). P'$$

**TA-LiRT** (by (139) and inv. of [TA-LiRT])

$$\Theta \cdot \Gamma \vdash_S s[p][q] \oplus m(s'[q']). P'$$

(by (139) and inv. of [TA-LiRT])

$$\Gamma_{\oplus} = \emptyset$$

We now have two cases, that we study in order to prove the existence of a suitable $\Gamma''$ such that $\Gamma'' \rightarrow_S \Gamma'''$:

- $p \in \text{senders}(\sigma)$. Then:
  
  $$\exists M \not\in \epsilon : \Gamma_0(s[p]) = M$$

  (by (139) and Prop.L.3) (144)

  $$\Gamma'_0 = \Gamma_0 \{ M \cdot q[m(S) \oplus \epsilon] / s[p] \}, \Gamma_2 \text{ and } \Theta \cdot \Gamma'_0 \vdash_{S_{\sigma}} s \triangleright \sigma \cdot (p, q, m(s'[q']))$$

  (by (139), (144), (142) and Prop.M.6) (145)

  $$\exists \Gamma''' = \Gamma_0, s[p] : S', \Gamma'_0 \text{ such that } \Gamma'' \rightarrow \Gamma'''$$

  (by (143), (145) and Def. D.4) (146)

  $$\Gamma'' \rightarrow_S \Gamma'''$$

  (by (143) and Def. F.1) (147)

  **Item 2.** Then:

  $$s[p] \notin \text{dom}(\Gamma_0)$$

  (by (139) and Prop.L.4) (148)

  $$\Gamma_0 = \Gamma_0, s[p] : q \oplus m(S) \cdot \epsilon, \Gamma_2 \text{ and } \Theta \cdot \Gamma_0 \vdash_{S_{\sigma}} s \triangleright \sigma \cdot (p, q, m(s'[q']))$$

  (by (139), (148), (142) and Prop.M.5) (149)

  $$\exists \Gamma''' = \Gamma_0, s[p] : S', \Gamma'_0 \text{ such that } \Gamma'' \rightarrow_S \Gamma'''$$

  (by (139), (141), (149) and Def. F.1) (150)

---

6Note that the same rewriting also allows to prove Prop.M.5, when $s[p] \notin \text{dom}(\Gamma)$. For clarity, we chose to keep Propositions M.5 and M.6 separate, with the rewriting in the proof of Prop.M.5 as simple as possible.
Therefore, for both cases above, using $\Gamma_\mu'$ from either (145) or (149), and $\Gamma'''$ from either (146) or (150), we obtain:

$$\text{a-safe}_G(\Gamma''') \quad \text{(by a-safe}_G(\Gamma), (143), Lemma M.2, (147)/(150) and Def. F.2, clause [SA→])}$$

$$\Theta \cdot \Gamma_\sigma' \vdash s[p]:S' \vdash P'_{\mu'} \quad \text{[TA-LrT]}$$

$$\Theta \cdot \Gamma_\sigma' \vdash s[p] \vdash (p, q, m(s'[q']))) \quad S_{\Theta} \cap S_{\sigma} = \emptyset \quad \text{[TA-]}$$

$$\exists \gamma': \Gamma \rightarrow_{S} \Gamma' \quad \text{and} \quad \Gamma' \subseteq \Gamma''' \quad \text{(by (143), (147)/(150) and Lemma M.3)}$$

$$\text{a-safe}_G(\Gamma') \quad \text{(by a-safe}_G(\Gamma), (153) and Def. F.2, clause [SA→])}$$

$$\Theta \cdot \Gamma' \vdash s[P] \vdash s[p] \vdash (p, q, m(s'[q']))) \quad \text{(by (152), (153), (154) and Lemma M.4)}$$

Hence, we conclude the proof by (153), (154) and (155).

**Item 2.** We have:

$$\Gamma = \Gamma_{\sigma}', \Gamma_{\sigma} \cdot \quad \text{and} \quad \Theta \cdot \Gamma_{\sigma}' \vdash s[p]:S \vdash P_{\mu'} \quad \Theta \cdot \Gamma_{\sigma} \vdash s[p] \vdash (p, q, m(s'[q']))) \quad S_{\Theta} \cap S_{\sigma} = \emptyset \quad \text{[TA-LrT]}$$

$$\Theta \cdot \Gamma_{\sigma}' \vdash s[p] \vdash (p, q, m(s'[q']))) \quad S_{\Theta} \cap S_{\sigma} = \emptyset \quad \text{[TA-]}$$

$$\Theta \cdot \Gamma_{\sigma}' \vdash s[p], q, m(S, S) \vdash \text{a-safe}_G(\Gamma) \quad \text{and clauses [SA→]/[SA-→] of Def. F.2)} \quad (160)$$

$$\Theta \cdot \Gamma_{\sigma}' \vdash s[p] \vdash (p, q, m(s'[q']))) \quad S_{\Theta} \cap S_{\sigma} = \emptyset \quad \text{[TA-}\sigma\text{]} \quad (159)$$

$$\exists k \in I : \quad m = m_k \quad \text{and} \quad S \subseteq S_k \quad \text{(by (156), (158), (159), a-safe}_G(\Gamma) \text{and clauses [SA→]/[SA-→] of Def. F.2)} \quad (160)$$

$$\Theta \cdot \Gamma_{\sigma}' \vdash s[p] \vdash (p, q, m(s'[q']))) \quad \text{[TA-}\sigma\text{]} \quad (159)$$

$$\exists \gamma_{\mu}' = \Gamma_{\mu}' \rightarrow_{S} \Gamma_{\mu}' \quad \text{and} \quad \Gamma' \subseteq \Gamma''' \quad \text{(by (156), (158), (159), [TA-sub] and Def. 2.6)} \quad (164)$$

$$\exists \gamma'' = \Gamma_{\mu}' \rightarrow_{S} \Gamma_{\mu}' \quad \text{(by (162), (159))} \quad (165)$$

We now have two cases, that we study in order to prove that $\Gamma'' \rightarrow_{S} \Gamma'''$:

- **$p \in \text{senders}(\sigma)$. Then:**
  
  $$\begin{align*}
  \Gamma'' & \rightarrow \Gamma''' \quad \text{(by (164), (165) and Def. D.4)} \quad (166) \\
  \Gamma'' & \rightarrow_{S} \Gamma''' \quad \text{(by (166) and Def. F.1)} \quad (167)
  \end{align*}$$

- **$p \notin \text{senders}(\sigma)$. Then:**
  
  $$\begin{align*}
  s[p] & \notin \text{dom}(\Gamma_{\sigma}') \quad \text{(by (159) and Prop.L.4)} \quad (168) \\
  \Gamma''(s[p]) & = S_k' \quad \text{(by (156), (158), (165) and (168))} \quad (169) \\
  \Gamma_{\sigma}(s[p]) & = q!m(S) \cdot \epsilon \quad \text{(by (159) and Fig.8)} \quad (170) \\
  \Gamma''(s[p]) & = (q!m(S) \cdot \epsilon) \cdot p \& i \in m(S_i) \cdot S_i' \quad \text{(by (164) and (170))} \quad (171) \\
  \Gamma'' & \rightarrow_{S} \Gamma''' \quad \text{(by (171), (169) and Def. F.1)} \quad (172)
  \end{align*}$$
Therefore, for both cases above, using either (167) or (172), we obtain:

\[
\begin{align*}
\text{a-safe}_S(\Gamma''') & \quad \text{by a-safe}_S(\Gamma), \ (164), \ Lemma \ M.2, \ (167)/(172) \ \text{and} \ Def. \ F.2, \ clause \ [\text{SA-} \text{a} \text{-} \text{in}] \tag{173} \\
\Theta \cdot \Gamma_k + P'_k[s'[q]/x_k] \quad & \quad \Theta \cdot \Gamma \vdash_S s \triangleright \sigma \quad \text{[TA-Lift]} \tag{174} \\
\Theta \cdot \Gamma'' \vdash_S P'_k[s'[q]/x_k] \quad & \quad \Theta \cdot \Gamma'_k \vdash_S s \triangleright \sigma \quad \text{[TA-]} \quad \text{(by (163), (156), (157), (159), (165), (173))} \tag{175} \\
\Gamma' : \Gamma \rightarrow S \Gamma' \quad & \quad \text{and} \quad \Gamma' \subseteq \Gamma'' \tag{176} \\
\text{a-safe}_S(\Gamma') & \quad \text{by a-safe}_S(\Gamma), \ (175) \ \text{and} \ Def. \ F.2, \ clause \ [\text{SA-} \text{a} \text{-} \text{in}] \tag{177} \\
\Theta \cdot \Gamma' \vdash_S P'_k[s'[q]/x_k] \quad & \quad s \triangleright \sigma \quad \text{by (174), (175), (176) \ \text{and} \ Lemma \ M.4} \tag{178}
\end{align*}
\]

Hence, we conclude the proof by (175), (176) and (177).

**Lemma F.5.** Let \( a\text{-safe}_S(\Gamma) \): then, \( a\text{-safe}_S(\Gamma) \) and if \( \Gamma = \Gamma', s[p], S \), then \( a\text{-safe}_S(\Gamma') \).

**Proof.** Assume \( a\text{-safe}_S(\Gamma) \): it means that all \( S \)-reductions of \( \Gamma \) are safe (by Def. F.2, clause \[\text{SA-} \text{a} \text{-} \text{in}] \). Note that, among such safe reductions of \( \Gamma \), there is also the subset of all its \( S \)-reductions (by Def. F.1): hence, by Def. F.2, we conclude \( a\text{-safe}_S(\Gamma) \) — which proves the first part of the statement.

For the second part of the statement, assume \( a\text{-safe}_S(\Gamma', s[p], S) \), and by contradiction, also assume that \( \Gamma' \) is not \( S \)-safe. Observe that by hypothesis and Def. F.2, \( \Gamma' \) can (possibly) map \( s[p] \) to a queue type, but not to a session type, nor to a session/queue type. Hence, by Def. F.1, \( \Gamma' \) cannot violate Def. F.2 due to new messages enqueued by \( s[p] \), nor due to messages sent to \( s[p] \). But then, the same violations of Def. F.2 can also be found by letting \( \Gamma' \), \( s[p], S \) reduce, which therefore is not \( S \)-safe — contradiction. Hence, we conclude \( a\text{-safe}_S(\Gamma) \).

**Proposition M.8.** Assume \( \Theta \cdot \Gamma, s[p] : M \vdash_S P \). Then, \( s \in S \) and \( P \equiv P_0 \) \( \vdash_S t \triangleright \sigma \).

**Proof.** By induction on the typing derivation, observing that the queue type can only be yielded by rule \[\text{TA-} \sigma \] (Fig. 8), which in turn requires \( s \in S \).

**Theorem F.6 (Asynchronous Subject Reduction).** Assume \( \Theta \cdot \Gamma \vdash_S P \) with \( \Gamma \) \( S \)-safe. Then, \( P \rightarrow P' \) implies \( \exists \Gamma' \) \( S \)-safe such that \( \Gamma \rightarrow \Gamma' \) \( \Gamma' \) and \( \Theta \cdot \Gamma' \vdash_S P' \).

**Proof.** By induction of the derivation of \( P \rightarrow P' \), and when the reduction holds by rule \[\text{R-Ctx} \], with a further structural induction on the reduction context \( C \). Most cases hold by inversion of the typing \( \Theta \cdot \Gamma \vdash_P \), and by applying the induction hypothesis.

The most complex cases are the base cases where \( P \rightarrow P' \) is due to rules \[\text{R-AOut} \] or \[\text{R-AIn} \], and messages are added or removed from the session queue. Such cases are proved in Lemma M.7 above.

In the inductive case where \( P = C[P_0] \) with \( C = C' \) \( \vdash Q \), we have:

\[
\begin{align*}
\exists \Gamma_1 : C'[P_0] & \rightarrow C'[P_1] \quad \text{and} \quad P' = C'[P_1] \vdash Q \ & \quad \text{(by inversion of \[\text{R-Ctx} \])} \tag{179} \\
S = S_1 \cup S_2 \quad & \quad \Theta \cdot \Gamma_1 \vdash_S C'[P_0] \quad \Theta \cdot \Gamma_2 \vdash_S Q \ & \quad \text{by (178), inv. \[\text{TA-} \]} \tag{180} \\
\Gamma = \Gamma_1, \Gamma_2 \ & \quad \text{such that} \quad S_1 \cap S_2 = \emptyset \quad \Theta \cdot \Gamma \vdash_S P \\
\text{a-safe}_S(\Gamma_1, \Gamma_2) & \quad \text{by \[\text{a-safe}_S(\Gamma), \ (179) \ \text{and} \ Lemma \ F.5\]} \tag{181}
\end{align*}
\]

Now, our intermediate goal is to prove:

\[
\text{a-safe}_S(\Gamma_1, \Gamma_2) \implies \text{a-safe}_S(\Gamma_1) \tag{182}
\]

For this purpose, we proceed by induction on \( \Gamma_2 \):

- base case \( \Gamma_2 = \emptyset \). Then, \( \Gamma_1 = \Gamma_1, \Gamma_2 \), and we conclude trivially by (180);
- inductive case \( \Gamma_2 = \Gamma_2', c : r \). Then, we have:

\[
\text{a-safe}_S(\Gamma_1, c : r) \quad \text{(by the induction hypothesis on (181))} \tag{183}
\]

and the following sub-cases:

- \( c = x \) (i.e., \( c \) is a variable). Then, observe that \( S_1 \)-safety (Def. F.2) only depends on typing context entries mapping channels with roles, and ignores any entry mapping variables: hence, from (182), we conclude \( \text{a-safe}_S(\Gamma_1) \);
- \( c = s[p] \). Then, we have the following possibilities:
We have thus proved (181), and therefore:

\[ \Gamma \text{ alternating queueing/dequeueing reductions give a corresponding synchronous reduction } \]

\[ \Gamma \text{ alternating queuing/dequeuing transitions (as in (188)) — which always possible by the hypothesis } \]

\[ \Gamma \text{ typing context that is the desired } \]

\[ \text{ and we conclude by (186) and (187).} \]

\[ \square \]

**Corollary F.7 (Async Type Safety).** If \( \emptyset \cdot \emptyset \vdash P \) and \( P \rightarrow^* P' \), then \( P' \) has no errors.

**Proof.** Similar to Cor. 4.9.

\[ \square \]

**Theorem 7.2.** If \( \varphi \) is decidable, then \( \Theta \cdot \Gamma \vdash G P \) with \( \varphi \) is decidable.

**Proof.** Similar to Thm. 4.11.

\[ \square \]

### N Asynchronous Typing Context Properties

**Lemma G.3.** For all \( \Gamma \), letting \( S = \{ s | \exists p : s[p] \in \text{dom}(\Gamma) \} \), we have:

\( 1 \) a-consistent(\( \Gamma \)) \( \iff \) a-safe\( G \) (\( \Gamma \));

\( 2 \) a-live\( G \) (\( \Gamma \)) \( \iff \) a-safe\( G \) (\( \Gamma \));

\( 3 \) a-live\( G \) (\( \Gamma \)) \( \iff \) a-df\( G \) (\( \Gamma \));

\( 4 \) a-nterm\( G \) (\( \Gamma \)) \( \iff \) a-df\( G \) (\( \Gamma \));

\( 5 \) a-consistent(\( \Gamma \)) \( \implies \) a-live\( G \) (\( \Gamma \));

\( 6 \) a-consistent(\( \Gamma \)) \( \wedge \) a-df\( G \) (\( \Gamma \)) \( \iff \) a-live\( G \) (\( \Gamma \));

\( 7 \) a-term\( G \) (\( \Gamma \)) \( \iff \) a-live\( G \) (\( \Gamma \));

\( 8 \) a-bound\( G \) (\( \Gamma \)) \( \iff \) a-bound\( G \) (\( \Gamma \));

\( 9 \) a-bound\( G \) (\( \Gamma \)) \( \iff \) a-safe\( G \) (\( \Gamma \)) \( \vee \) a-df\( G \) (\( \Gamma \));

\( 10 \) a-live\( G \) (\( \Gamma \)) \( \iff \) a-live\( G \) (\( \Gamma \)) \( \iff \) a-live\( G \) (\( \Gamma \)).

**Proof.** The negated implications in the statement are proved in Ex.G.4. The remaining implications are similar to those in Lemma 5.9, using the corresponding asynchronous definitions.

\[ \square \]

**Lemma N.1.** Assume \( \text{dom}(\Gamma) = \{ s \} \) and \( \text{live}(\Gamma) \). Then, for all synchronous (i.e., queue-less) typing contexts \( \Gamma' \):

\( 1 \) \( \Gamma \rightarrow^* \Gamma' \) implies \( \Gamma \rightarrow_{\{s\}} \Gamma' \);

\( 2 \) \( \Gamma \rightarrow_{\{s\}} \Gamma'' \) implies \( \exists \Gamma'' : \Gamma'' 

\rightarrow_{\{s\}} \Gamma' \) and \( \Gamma \rightarrow^* \Gamma' \).

**Proof. (Item 1)** We first prove the statement for one reduction step, i.e.:

\[ \forall \Gamma, \Gamma' : \Gamma \rightarrow \Gamma' \implies \Gamma \rightarrow_{\{s\}} \Gamma' \quad (188) \]

where the two-step reduction represents a message being queued, and then immediately consumed — which always possible by the hypothesis \( \text{live}(\Gamma) \). Then, we prove the main statement by induction on the number of reductions in \( \Gamma \rightarrow \Gamma' \), using (188) for the inductive step.

**Item 2** Consequence of [Denielou and Yoshida 2013, Thm 4.1] and [Bocchi et al. 2015, Thm. 6], that say (roughly): if \( \Gamma \) is live, then queued outputs are eventually consumed, and external choices are eventually triggered. More in detail: if \( \Gamma'' \) contains queued messages, we can let \( \Gamma'' \) reduce by consuming each queued message, thus reaching a queue-less typing context that is the desired \( \Gamma' \); then, we can reorder the transitions in \( \Gamma \rightarrow_{\{s\}} \Gamma'' \rightarrow_{\{s\}} \Gamma' \) into a sequence of alternating queueing/dequeueing transitions (as in (188)) — which always possible by the hypothesis \( \text{live}(\Gamma) \); the resulting alternating queueing/dequeueing reductions give a corresponding synchronous reduction \( \Gamma \rightarrow^* \Gamma' \).

\[ \square \]
Lemma N.2. Assume \( \text{dom}(\Gamma) = \{s\} \) and \( \text{live}(\Gamma) \). Then, \( a\text{-live}_{\{s\}}(\Gamma) \).

Proof. Let us define \( \phi = a\text{-beh}_{\{s\}}(\Gamma) \) similarly to Def. K.1, but using asynchronous reductions \( \rightarrow_{S} \). Then, we prove that \( \phi \) is an asynchronous liveness property, by using Lemma N.1(2) to show that active external choices can be triggered (clause \([\text{LA-\&}]\) of Def. G.2(4)), and queued messages can be consumed (clause \([\text{LA-!}]\) of Def. G.2(4)). Finally, by Def. G.2(4), we conclude \( a\text{-live}_{\{s\}}(\Gamma) \).

□

Lemma N.3. Assume \( \text{dom}(\Gamma) = \{s\} \) and \( \text{live}^+ (\Gamma) \). Then, \( a\text{-live}^+_{\{s\}}(\Gamma) \).

Proof. We use \( \phi \) as in the proof of Lemma N.2, that we know is an \( \{s\} \)-liveness property by Lemma 5.9(7) and Lemma N.2. In order to prove the “moreover…” clauses of \( \{s\} \)-liveness*, we also use the following result, that follows by Lemma N.1(2):

Proposition N.4. If \( \text{live}(\Gamma) \) and \( \Gamma \rightarrow_{\{s\}} \Gamma_0, \Gamma_1 \) and \( \Gamma_0 \rightarrow_{\{s\}} \Gamma \),

there are \( \Gamma_0', \Gamma_1' \) queue-less, such that \( \Gamma_0 \rightarrow_{\{s\}} \Gamma_0' \rightarrow_{S} \Gamma_0' \) and \( \Gamma_1 \rightarrow_{\{s\}} \Gamma_1' \rightarrow_{S} \Gamma_1' \) and \( \Gamma \rightarrow_{S} \Gamma_0', \Gamma_1' \)
i.e., if \( \Gamma \) is live and produces a loop under asynchronous semantics, then it also produces a corresponding loop under synchronous semantics.

Then, to prove the “moreover…” parts of clauses \([\text{LA-\&}^+] / [\text{LA-!}^+]\) of Def. G.2(5), we proceed by contradiction, similarly to the proof of Thm. K.15: we assume that there is no asynchronous traversal set that satisfies the requirements of Def. G.2(5) (similarly to step (113)); this means that \( \Gamma \) can perform asynchronous reduction loops that, by Prop. N.4, imply the existence of corresponding synchronous reduction loops, whose form matches the one described in steps (114)–(116). This leads to the contradiction that \( \Gamma \) is not \( \text{live}^+ \). Therefore, clauses \([\text{LA-\&}^+] / [\text{LA-!}^+]\) of Def. G.2(5) hold, and we conclude \( a\text{-live}^+_{\{s\}}(\Gamma) \).

□