Less Is More: Multiparty Session Types Revisited

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

ACM Reference Format:

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

Authors’ addresses: Alceste Scalas, Imperial College London, UK, alceste.scalas@imperial.ac.uk; Nobuko Yoshida, Imperial College London, UK, n.yoshida@imperial.ac.uk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/1-ART30
https://doi.org/10.1145/3290343

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:

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 \( a \):

\[
G = s \rightarrow c : \{ \text{login}.c \rightarrow a \text{: passwd(Str)}.a \rightarrow s \text{: auth}(\text{Bool}).\text{end}, \text{cancel}.c \rightarrow a \text{: quit}.\text{end} \} \tag{1}
\]

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 \( G \):

\[
\begin{align*}
S_s &= c @ \{ \text{login}.a \& \text{auth}(\text{Bool}), \text{cancel} \} \\
S_c &= s @ \{ \text{login}.a @ \text{passwd}(\text{Str}), \text{cancel}.a @ \text{quit} \} \\
S_a &= a @ \{ \text{passwd}(\text{Str}), s @ \text{auth}(\text{Bool}), \text{quit} \}
\end{align*}
\tag{2}
\]

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:

\[
\Gamma \vdash P \quad \text{and} \quad P \rightarrow P' \quad \text{implies} \quad \exists \Gamma' : \Gamma' \vdash P'
\tag{3}
\]

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 \( c_s, c_r, c_a \) 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 \( s @ n(\text{Int}).\text{end} \), and they could type a parallel process \( P = P_1 \parallel P_2 \), where \( P_1 \) and \( P_2 \) interact according to the types, with \( P_1 \) sending a message \( m(\text{"Hello")} \) (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 [Coppo 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:

$$\Gamma \vdash P \text{ with } \Gamma \text{ consistent and } P \rightarrow P' \implies \exists \Gamma' \text{ consistent: } \Gamma \rightarrow^{\ast} \Gamma' \text{ and } \Gamma' \vdash P' \quad (4)$$

(where $\Gamma \rightarrow^{\ast} \Gamma'$ 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.
2. We uncover that the resulting severe limitations lead to subtle theoretical issues (§3).
3. 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.
4. 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).
5. 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).
6. 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.
2 MULTIPARTY SESSION TYPES

This section describes the multiparty session $\pi$-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 $\pi$-Calculus

The multiparty session $\pi$-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 $\pi$-calculus syntax is defined as follows:

$\begin{align*}
c, d & ::= x \mid s[p] & \text{(variable, channel with role p)} \\
P, Q & ::= 0 \mid P \mid Q \mid (vs) P & \text{(inaction, composition, restriction)} \\
c[q] \sum_{i \in I} m_i(x_i).P_i & \text{ (branching from role q with I \neq \emptyset)} \\
def D \ in \ P & \mid X(\overline{c}) \mid \text{err} & \text{(process definition, process call, error)} \\
D & ::= X(\overline{c}) = P & \text{(declaration of process variable X)}
\end{align*}$

Restriction, branching and declarations act as binders, as expected; $fc(P)$ is the set of free channels with roles in $P$, and $fv(P)$ is the set of free variables in $P$. We adopt a form of Barendregt convention: bound sessions and process 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 $(vs) P$ declares a new session $s$ with scope limited to process $P$. Process $c[q] \oplus m(d).P$ performs a selection (internal choice) towards role $q$, using the channel $c$: the message label $m$ is sent with the payload channel $d$, 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 $\text{def } X(\overline{c}) = P \ in \ Q$ 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 $C$ is: $C ::= C \mid P \mid (vs) C \mid \text{def } D \ in \ C \mid []$
Fig. 1. MPST π-calculus semantics, defined up-to standard structural congruence [Scalas and Yoshida 2018a].

Reduction \( \rightarrow \) is inductively defined in Fig. 1, up-to a standard structural congruence \( \equiv \) [Scalas and Yoshida 2018a] including \( \alpha \)-conversion. We say that \( P \) has an error if, for some \( C, P = C[\text{err}] \).

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-Era] says that a parallel composition of mismatching selection and branching processes reduces to \texttt{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.

\[
\begin{align*}
\text{(vs) } (P_s \mid P_c \mid P_a) & \quad \text{where: } \\
& \begin{cases}
(P_s = s[s][c]@cancel) \\
(P_c = s[c][s] \sum\{ \text{login}.s[c][a]@\text{passwd}("XYZ") , \text{cancel}.s[c][a]@\text{quit} \\
(P_a = s[a][c] \sum\{ \text{passwd}(y).s[a][s]@\text{auth}(y = "secret") , \text{quit} \\
\end{cases}
\end{align*}
\]

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 \( \text{login} \) or \( \text{cancel} \) from \( s \); then, in the first case it uses \( s[c] \) to send \( \text{passwd} \) to \( a \); in the second case, it uses \( s[c] \) to send \( \text{quit} \) to \( a \). By Def. 2.2, we have the reductions:

\[
\text{(vs) } (P_s \mid P_c \mid P_a) \rightarrow (\text{vs) } (0 \mid s[c][a]@\text{quit} \mid P_a) \rightarrow (\text{vs) } (0|0|0) \equiv 0
\]

2.2 Types, Subtypes, and Typing

Session types (Def. 2.4) describe the intended use of communication channels in the MPST π-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:

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

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) \( p@_{i \in I} m_i(S_i).S_i' \), 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 \(S'\). 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 that recursive types are contractive: i.e., in \(\mu t S\) 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., \(p \nbar m\) stands for \(p \nbar m(\text{end}).\) 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:

\[
\forall i \in I \quad S_i \leq T_i \quad S'_i \leq T'_i \quad \text{[Sub-\&]} \quad \forall i \in I \quad T_i \leq S_i \quad S'_i \leq T'_i \quad \text{[Sub-\&]} \\
\text{end} \leq \text{end} \quad \text{[Sub-end]} \\
S[\mu t S|t|] \leq T \quad \text{[Sub-\&L]} \\
S \leq T[\mu t T|t|] \quad \text{[Sub-\&R]}
\]

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:

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

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

We write \(s \in \Gamma\) iff \(\forall p : s[p] \notin \text{dom}(\Gamma)\) (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 \(\Gamma \leq \Gamma'\) iff \(\text{dom}(\Gamma) = \text{dom}(\Gamma')\) and \(\forall c \in \text{dom}(\Gamma') : \text{dom}(c) \leq \text{dom}(c')\).

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:

\[
\Theta \cdot \Gamma \vdash P \quad (\text{with } \Theta \text{ omitted when empty}) \quad (5)
\]

meaning: “given the process types in \(\Theta, \Gamma\) 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], \(\Theta \vdash X:S_1, \ldots, S_n\) holds if \(\Theta\) maps \(X\) to an \(n\)-tuple of types \(S_1, \ldots, S_n\). By [T-Str], \(\Gamma \vdash c:S\) holds if \(\Gamma\) only contains one entry \(c:S\) with \(S \leq S'\): i.e., when typing processes, [T-Str] 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-Str] makes the type system equi-recursive [Pierce 2002, Ch. 21]. Finally, end(\(\Gamma\)) holds if \(\Gamma\)’s entries are end-typed (under [T-Str]).

The other rules in Fig. 2 define the process typing judgement in (5). The termination rule [T-\(0\)] says that \(0\) is typed if all channels in \(\Gamma\) are end-typed. By the process definition rule [T-def], def \(X(x) = P\) in \(Q\) is typed if \(P\) uses the arguments \(x_1, \ldots, x_n\) according to \(S_1, \ldots, S_n\), and the latter

---

1Our \(\leq\) 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].
is the type of $X$ when typing both $P$ and $Q$: this means that $P$ can refer to $X$, and this allows to type recursive processes. By the process call rule $[T\cdot\_X]$, $X(c)$ is typed if the types of $c$ 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 $[T\cdot\&]$, $c[q]\sum_{i\in I}m_i(y_i).P_i$ 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 $[T\cdot\&]$ says that $c[q]\oplus m(d).P$ is typed if $c$ has type $S$, where $S$ is an internal choice towards $q$ with message label $m$. By the parallel rule $[T\cdot\|]$, two parallel processes are typed by splitting the context in the premises. The session restriction rule $[T\cdot\nu]$ 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:

\[
\begin{align*}
\vdash [s[s] : S_s + P_s] & \vdash [s[c] : S_c + P_c] \quad [T\|-] \quad \vdash [s[s] : S_s, [s[c] : S_c + P_c \mid P_c] \mid P_a] \quad [T\|-] \\
\vdash [s[a] : S_a + P_a] & \quad \text{where } \Gamma = s[s] : S_s, [s[c] : S_c, s[a] : S_a]
\end{align*}
\]

The process $P_s \mid P_c \mid P_a$ is typed by rule $[T\|-]$, 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 $[T\cdot\&][T\cdot\&]$: 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 $\leq$ (to cater for subtyping), and we specify transition labels for inputs, outputs, and communication.

Definition 2.8. Let $\alpha$ have the form $s[p\&q:m(S))$, or $s[p\&q:m(S))$, or $s[p\&q:m(S)$ (for any roles $p, q$, message label $m$, and type $S$). The typing context transition $\rightarrow$ is inductively defined by the rules:
We write \( \Gamma \xrightarrow{\alpha} \Gamma' \) iff \( \Gamma \xrightarrow{\alpha} \Gamma' \) for some \( s, p, q, m \). We write \( \Gamma \rightarrow \) iff \( \Gamma \rightarrow \Gamma' \) for some \( \Gamma' \), and \( \Gamma 
rightarrow \) 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 \([\Gamma-\Theta]/[\Gamma-\&]\) in Def. 2.8, a typing context entry can transition to one of its continuations by firing an output label of the form \( s \cdot p \cdot q \cdot m(S) \) (in case of selection types), or an input label of the form \( s \cdot p \& q \cdot m(S) \) (in case of branching types). Rule \([\Gamma-\text{COMM}]\) models type-level communication: e.g., it allows two entries \( s[p] \cdot q : S_p, s[q] : S_q \) 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 \( S_q \). When all such conditions hold, \( s[p] \cdot p \cdot q \cdot m \cdot S \) transition to the respective continuations, by firing a communication label \( s \cdot p \cdot q \cdot m \) 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 reprinted in \S 5.

2.3 Towards Subject Reduction and Type Safety

In \S 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:

\[
s[p] \cdot q \odot \text{foo}(\text{end}), \ s[q] \cdot p \& \text{bar}(\text{end}), \ s'[r] \cdot \text{end} \vdash s[p][q] \odot \text{foo}(s'[r]) | s[q][p] \sum \text{bar}(x) \rightarrow \text{err} \quad (6)
\]

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 \([\text{T-v}]\) 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 countereexample to type-safety, where \( \Gamma \) is the context in \( (6) \):

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

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

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

- \textbf{(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;
- \textbf{(D2)} \( \varphi \) must be preserved when the typing rule \([\text{T-l}]\) splits typing contexts (see derivation in Ex. 2.7);
- \textbf{(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 \([\text{T-v}]\), to correctly type session restriction \( (\nu s) \text{ P} \), and handle cases like \( (7) \).
In §4, we show how our new MPST theory establishes its foundations, and ϕ 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 [2018b]: 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 \( \varphi = \text{consistent} \). This yields the classic session restriction typing rule:

\[
\frac{\Gamma = [s[p]:S_p]_{p \in \Gamma} \quad s \not\in \Gamma \quad \text{consistent}(\Gamma') \quad \Theta \cdot \Gamma, \Gamma' \vdash P}{\Theta \cdot \Gamma \vdash (\nu s:\Gamma') P} \quad [\text{T-vCLASSIC}]
\]

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 \( (\nu s) \ldots \) in (7): hence, with rule [T-vCLASSIC], 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-vCLASSIC] above, we might try to type our opening example as:

\[
\frac{\Gamma \text{ consistent} \quad \vdash \nu s: (\nu c: P_s | P_c | P_a) \quad \text{[T-\text{I}]}}{\Theta \vdash (\nu s: \Gamma) \quad \text{[\nu s:\Gamma]}} \quad \text{[T-vCLASSIC]}
\]

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-vCLASSIC] does not hold. To see why, we need to check all pairs of types for session \( s \):

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

Intuitively, \( S_s | a \) 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 err).

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 \([\text{T-vClassicG}]\) below, that seemingly fixes derivation (8):

\[
\Gamma' = [s[p]: G|p]_{\text{proroles}(G)} \
\Theta \cdot \Gamma' \vdash P \\
\Theta \cdot \Gamma \vdash (\text{vs} : \Gamma') P \\
\]  

\([\text{T-vClassicG}]\)

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. 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(Str)}. a \rightarrow s: \text{auth(Bool)}\}
\]

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 & \text{passwd(Str). s}@\text{auth(Bool)}\}
\]

We now have two possibilities, depending on how we choose the merging operator \(\cap\) (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 \([\text{T-vClassicG}]\) with "full merging" projections breaks subject reduction proofs. E.g., take \(P\) typed by \(\Theta \cdot \Gamma \vdash P\) with \(P = (\text{vs} : \Gamma) P_0 \rightarrow (\text{vs} : \Gamma') P_1 = P'\) (induced by \(P_0 \rightarrow P_1\) and rule \([\text{R-Ctx}]\) in Fig. 1).

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
(1) OAuth2 fragment. (See global type (1) in §1 | (See types (2) in §1, and Γ in Ex.2.7)

(2) Recursive two-buyers protocol. This is a mild variation of a typical example in MPST literature. Alice (a) queries the store (s) for an item, and the store replies with a price; then, she asks Bob (b) to split the price. If he says yes, then she buys the item from the store; if he says no, then Alice recursively retries, proposing another split to Bob; at any point, Alice can cancel her bargaining with Bob, and say no to the store.

\[
\begin{align*}
\text{s}[\text{a}]: & s@\text{query(Str).s&price(Int).µt.b@\{\text{split(Int).b&\{yes,s@buy.end.\} cancel.s@no}\}} \\
\text{s}[\text{s}]: & a@\text{query(Str).a&price(Int).a&\{\text{buy.end, no.end}\}} \\
\text{s}[\text{b}]: & µt.a&\{\text{split(Int).a&\{yes.end, no.t\} cancel.end}\}
\end{align*}
\]

(3) Recursive map/reduce. The mapper (m) sends a datum to n workers (w_i, ..., w_n, for some given n), and each one sends a result to the reducer (r); then, the reducer tells the mapper whether to continue with another iteration, or stop: in the first case, the mapper loops, while in the second case, it stops the workers.

\[
\begin{align*}
\text{s}[\text{a}]: & s@\text{query(Str).s&price(Int).µt.b@\{\text{split(Int).b&\{yes,s@buy.end.\} cancel.s@no}\}} \\
\text{s}[\text{s}]: & a@\text{query(Str).a&price(Int).a&\{\text{buy.end, no.end}\}} \\
\text{s}[\text{b}]: & µt.a&\{\text{split(Int).a&\{yes.end, no.t\} cancel.end}\}
\end{align*}
\]

(4) Independent multiparty workers. The starter process (s) sends a datum to n worker processes (w_a, ..., w_n, for some given n), and each one starts exchanging datum/result messages with two other workers (w_b and w_c, for i in [1, n]). Each triplet of workers w_a, w_b, and w_c (i in [1, n]) keeps interacting until w_a sends stop to w_b, who forwards stop to w_c.

\[
\begin{align*}
\text{s}[\text{a}]: & s@\text{query(Str).s&price(Int).µt.b@\{\text{split(Int).b&\{yes,s@buy.end.\} cancel.s@no}\}} \\
\text{s}[\text{s}]: & a@\text{query(Str).a&price(Int).a&\{\text{buy.end, no.end}\}} \\
\text{s}[\text{b}]: & µt.a&\{\text{split(Int).a&\{yes.end, no.t\} cancel.end}\}
\end{align*}
\]

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 §5). 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. $\varphi$ is a safety property of typing contexts iff:

\[
\begin{align*}
[S,\cdot] & \quad \varphi(\Gamma, s[p]; q@j \in m_j(S_j) \cdot S_j', s[q]; p@j \in m_j(T_j) \cdot T_j') \quad \text{implies} \quad I \subseteq J, \quad \text{and} \quad \forall i \in I : S_i \leq T_i; \\
[S,\mu] & \quad \varphi(\Gamma, s[p]; \mu t.S) \quad \text{implies} \quad \varphi(\Gamma, s[p]; S(\mu t.S/\iota)); \\
[S,-] & \quad \varphi(\Gamma) \quad \text{and} \quad \Gamma \rightarrow \Gamma' \quad \text{implies} \quad \varphi(\Gamma').
\end{align*}
\]

We say $\Gamma$ is safe, written $\text{safe}(\Gamma)$, if $\varphi(\Gamma)$ for some safety property $\varphi$.

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 $[S,\cdot\&]$ 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 $[S,\mu]$ says that $\varphi$ contains all recursive type unfoldings: this allows rule $[S,\cdot\&]$ to check unfolded types, where $@/\&$ occur at the the top-level. By rule $[S,-]$, 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 $\varphi$ as $\varphi = \{\Gamma' \mid \Gamma \rightarrow^* \Gamma'\}$, i.e., containing $\Gamma$ and all its reductions; (2) checking that $\varphi$ is a safety property, because all its elements satisfy the clauses of Def. 4.1; and (3) concluding that, since $\varphi(\Gamma)$ holds, $\Gamma$ is safe. Instead, the typing context in (6) is not safe: any property $\varphi$ containing such typing context is not a safety property, as it violates clause $[S,\cdot\&\&]$ of Def. 4.1.

Def. 4.1 also has the properties in Lemma 4.3 below, useful for proving subject reduction: typing context splits preserve safety (item 1, which satisfies the remaining desideratum (D2) in §2.3); if $\Gamma$ is safe, then supertype/reductions commute (item 2); supertypepreserves safety (item 3).

Lemma 4.3. For all typing contexts $\Gamma$ and $\Gamma'$:

1. if $\Gamma, \Gamma'$ is safe, then $\Gamma$ is safe;
2. if $\Gamma$ safe and $\Gamma \leq \Gamma' \rightarrow \Gamma''$ (for some $\Gamma''$), then there is $\Gamma'''$ such that $\Gamma \rightarrow \Gamma'' \leq \Gamma''$;
3. if $\Gamma$ is safe and $\Gamma \leq \Gamma'$, 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 4.4 (General Multiparty Session Type System). The general MPST typing judgement is inductively defined by the rules in Fig. 2 — with rule $[T,\cdot\&]$ restricted as follows:

\[
\frac{\Gamma' = [s[p]: p@E] \varphi(\Gamma') \quad s \notin \Gamma \quad \Theta \cdot \Gamma, \Gamma' + P \quad \Theta \cdot \Gamma + (\nu s; \Gamma') P}{\Theta \cdot \Gamma + P \quad \text{TGen-}\varphi} \quad \text{where} \quad \varphi \quad \text{is a safety property}
\]

Given a safety property $\varphi$, we write “$\Theta \cdot \Gamma + P$ with $\varphi$” to instantiate $\varphi$ in $[TGen-\varphi]$ above; when “with $\varphi$” is omitted, then the instantiation is $\varphi = \text{safe}$ (i.e., the largest safety property, cf. Def. 4.1).
Example 4.5. Take the (wrong) typing derivation (8) in §3.1, and replace the (wrong) application of rule [TGen-vCLASSIC] with [TGen-v] from Def. 4.4, instantiating \( \varphi \) with the safety property of Ex. 4.2 (that contains \( \Gamma \)). The resulting typing derivation is correct.

Ex. 4.5 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. 4.4) and the classic one (§3.1) is that [TGen-v] uses a (parametric) safety property \( \varphi \), instead of consistency.\(^2\) 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. 4.6, based on Lemmas 4.3 to 4.3); from this, we get that typed processes “never go wrong” (Cor. 4.7). And again, unlike classic MPST, these results are not limited by consistency.

Theorem 4.6 (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 4.7 (Type safety). If \( \cdot \Theta \vdash P \) 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. 4.6, 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. 4.7 as a special case of the result above, with \( \Theta = \emptyset \) and \( \Gamma = \Gamma' = \emptyset \) (that is vacuously safe).

Example 4.8. Take our opening example in §1, and its typed process from Ex. 2.7 and 4.5. Using our new Thm. 4.6 instead of the classic MPST subject reduction (4) in §1, we infer that all process reductions are well-typed. And by Cor. 4.7, we are guaranteed that they do not contain errors.

Finally, note that type checking is decidable, whenever Def. 4.4 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. 4.6 and Cor. 4.7 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. 4.4: subject reduction and type safety hold for any safety property \( \varphi \).

Theorem 4.9. 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. 4.4), 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. 4.6, Cor. 4.7), 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 \( p \)):

\[
\begin{align*}
\mathit{s}[p];q &\parallel m_1, r \parallel m_2, \mathit{s}[q];r &\parallel m_3, p \parallel m_1, \mathit{s}[r];p &\parallel m_2, q \parallel m_3
\end{align*}
\]

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. 4.4 to enforce them, without consistency limitations.

\(^2\)In §5.4, we show that all typing derivations of classic MPST are valid under Def. 4.4: consistency implies safety, hence in [TGen-v] we can let \( \varphi \) be 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 \( P \rightarrow P' \not\equiv \emptyset \). P is terminating iff it is deadlock-free, and \( \exists j \) finite such that, for all \( n \geq j \), \( P = P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_n \) implies \( P_n \equiv \emptyset \). P is never-terminating iff \( P \rightarrow P' \) implies \( P' \not\equiv \emptyset \). P is live iff \( P \rightarrow P' \equiv \mathbb{C}[Q] \) implies:

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

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

3. 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 \( P_j' \rightarrow C''[Q''] \) (for some \( C'' \));
4. 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 \( P_j' \rightarrow C''[Q_k'(s'[r]/x_k)] \) (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 \( \emptyset \); \( P \) is terminating when it always reaches \( \emptyset \) 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:

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

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 \( Q \):

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

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:

\[
Q = s[r][q]@m_1.s[r][q]@m_2.0 \mid Q' \quad \text{where} \quad Q' = \left\{ \begin{array}{ll}
\text{def } Z(z) = z[r'][r'] \oplus m_3, Z(z) \in Z(s[r']) & \text{in } Z(s[r']) \\
\text{def } Z'(z') = z'[r'[r'] \oplus m_3(x).Z'(z') \in Z'(s[r']) & \text{in } Z'(s[r'])
\end{array} \right.
\]

i.e., \( r \) sends \( m_1 \) and then \( m_2 \) to \( q \), and this causes \( q \) to send \( \text{resp} \) 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.\footnote{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.}

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) 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 $\text{def } X(x_1:S_1, \ldots, x_n:S_n) = Q$ in $P'$, for all $i \in 1..n$, $S_i \not\in \text{end}$ implies that a call $Y(..., x_i, ...)$ can only occur in $Q$ as subterm of $x_i[q]\sum_{j \in J} m_j(y_j).P_j$ or $x_i[q]@m(c).P''$ (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) $fv(P) = \emptyset$; (iii) $\Gamma = \emptyset$, $s[p]:S$ with $S \not\in \text{end}$ and $\text{end}(\Gamma_0)$; (iv) in all subterms $(\nu s':\Gamma')P'$ of $P$, we have $\text{end}(\Gamma')$.

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, $P \equiv \bigcap_{p \in P} P_p$ and each $P_p$ either is $\emptyset$ (up-to $\equiv$), or only plays $p$ in $s$. Then, $\Gamma \rightarrow$ implies $\exists \Gamma', P'$ such that $\Gamma \rightarrow \Gamma'$, $P \rightarrow^* P'$ and $\emptyset \cdot \Gamma' \vdash P'$, where $P' \equiv \bigcap_{p \in P} P'_p$ 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 $[\tau\cdot]q$ in Fig. 2, and subtyping). This observation will be a crucial when reasoning about process liveness ($\S$ 5.5). Also note that Thm. 5.4 relies on item 1 of Def. 5.3. In fact, by rule $[\tau\cdot\text{def}]$ (Fig. 2), an unguarded definition $X(x:S) = X(x)$ can be typed with any $S$; therefore, we have e.g.:

$\emptyset \cdot s[p]:q@m, s[q]:p&m \vdash \text{def } X(x:q@m) = X(x) \text{ in } X(s[p]) \mid s[q]p \sum m$

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 $\S$ 5.5, we will show how they can statically enforce the run-time process properties discussed in $\S$ 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.
\[ (1) \ \text{Γ is safe, written safe(Γ), iff:} \]
\[ \Gamma \vdash \forall s, p, q, m, m', S, S'. \]
\[ (\text{see Def. 4.1}) \]
\[ (s; p@q(m(S)) \top \land (s; p@q(m'(S')) \top \Rightarrow (s; p@q(m) \top) \land [s; p@q(m)] Z) \]

\[ (2) \ \text{Γ is deadlock-free, written df(Γ), iff:} \]
\[ \Gamma \rightarrow \Gamma' \land \neg \Rightarrow \text{implies } \text{end}(\Gamma') \]
\[ \Gamma \vdash \forall s, p, q, m, [s; p@q(m)] \bot \land (s; p@q(m) \top) \land [s; p@q(m)] Z \]

\[ (3) \ \text{Γ is terminating, written term(Γ), iff:} \]
\[ \Gamma \rightarrow \Gamma_0 \land \neg \Rightarrow \text{implies } \text{end}(\Gamma_0) \]
\[ \Gamma \vdash \forall s, p, q, m, [s; p@q(m)] \bot \land (s; p@q(m) \top) \land [s; p@q(m)] Z \]

\[ (4) \ \text{Γ is never-terminating, written nterm(Γ), iff:} \]
\[ \Gamma \rightarrow \Gamma' \land \neg \Rightarrow \text{implies } \text{end}(\Gamma') \]
\[ \Gamma \vdash \forall s, p, q, m, (s; p@q(m) \top) \land [s; p@q(m)] Z \]

\[ (5) \ \text{Γ is live, written live(Γ), iff:} \]
\[ \varphi(\Gamma), \text{for some } \varphi \text{ such that} \]
\[ \text{[L-ε]} \ \text{whenever } \varphi(\Gamma), s[p]:S \text{ with } S = q \land \exists i \in \mathbb{N} \land |I| \text{ (Def. 5.5) such that, } \forall \Gamma_i \in \mathbb{N} \land |I| \text{ we have } \Gamma_i = \Gamma'' \land s[p]:S'' \text{ (for some } \Gamma'', i \in I) \]
\[ \text{[L-η]} \ \text{when taking } \varphi(\Gamma), \text{ above, moreover, } \Gamma', s[p]:S \text{ belongs to some fair} \]
\[ \text{traversal set } \mathcal{X} \text{ with targets } \mathcal{Y} \text{ (Def. 5.5) such that, } \forall \Gamma_i \in \mathbb{N} \land |I| \text{ we have } \Gamma_i = \Gamma'' \land s[p]:S'' \text{ (for some } \Gamma'', i \in I) \]

\[ (6) \ \text{Γ is live*, written live*(Γ), iff:} \]
\[ \varphi(\Gamma), \text{ for } \varphi \text{ such that} \]
\[ \text{[L-ε*]} \ \text{clause } [L-ε] \text{ above; moreover, } \Gamma', s[p]:S \text{ belongs to some fair} \]
\[ \text{traversal set } \mathcal{X} \text{ with targets } \mathcal{Y} \text{ (Def. 5.5) such that, } \forall \Gamma_i \in \mathbb{N} \land |I| \text{ we have } \Gamma_i = \Gamma'' \land s[p]:S'' \text{ (for some } \Gamma'', i \in I) \]

\[ (7) \ \text{Γ is live**, written live**(Γ), iff:} \]
\[ \varphi(\Gamma), \text{ for } \varphi \text{ such that} \]
\[ \text{[L-ε**]} \ \text{clause } [L-ε] \text{ above; moreover, } \Gamma', s[p]:S \text{ belongs to some fair} \]
\[ \text{traversal set } \mathcal{X} \text{ with targets } \mathcal{Y} \text{ (Def. 5.5) such that, } \forall \Gamma_i \in \mathbb{N} \land |I| \text{ we have } \Gamma_i = \Gamma'' \land s[p]:S'' \text{ (for some } \Gamma'', i \in I) \]

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 \([\cdot\cdot\cdot]\) of Fig. 5(5), if \( S \) is an external choice, then \( \Gamma \) can reduce until some branch of \( S \) is triggered; and by clause \([\cdot\cdot\cdot]\), 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 rules 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:

\[
\Gamma \in \mathcal{Y} \quad \Rightarrow \quad \exists s, p, q : \quad \exists m : \quad \Gamma \xrightarrow{s,p,q,m} \quad \forall m : \quad \Gamma \xrightarrow{s,p,q,m} \Gamma' \quad \text{implies} \quad \Gamma' \in \mathcal{X}
\]

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 reduction belongs to \( \mathcal{X} \) (clause “…\( \Gamma' \in \mathcal{X} \)”). 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:

\[
\Gamma = s[p]:\mu t.q@\{m_1.t, m_2\}, s[q]:\mu t.p&\{m_1.t, m_2.r@m_3\}, s[r]:q&m_3
\]

\[
\Gamma' = s[p]:\alpha d, s[q]:\alpha d, s[r]:\alpha d
\]

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 \( m_1 \), yielding the infinite run \( \Gamma \xrightarrow{s,p,q,m_1} \Gamma \xrightarrow{s,p,q,m_1} \cdots \); 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 by “unfair scheduling.” E.g., consider:

\[
\Gamma = s[p]:q&m_1.q@\{m_2, s[q]:p&m_2, s[r]:r@\{m_2.t, s[r']:r.t&m_2.t
\]

\[
\Gamma' = s[p]:\alpha d, s[q]:\alpha d, s[q']:\alpha d, s[r]:\mu t.r@\{m_2.t, s[r']:\mu t.r&m_2.t
\]

Note that \( \Gamma \) is live, and \( \Gamma' \) is reachable from \( \Gamma \), via the reductions \( \Gamma \xrightarrow{s,p,q,m_1} \Gamma \xrightarrow{s,p,q,m_1} \cdots \); however, \( \Gamma' \) can be infinitely delayed in the unfair run \( \Gamma \xrightarrow{s,r,r'm_2} \Gamma \xrightarrow{s,r,r'm_2} \cdots \) 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 \( r, r' \).

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 \( \Gamma_f \) where such input/output has been fired, by interacting with a matching output/input.
Table 1. Verification of the multiparty protocols in Fig. 4 against the properties in Fig. 5. The results for protocol (3) hold for n ≥ 1, while the results for protocol (4) hold for n ≥ 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>false</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>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

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 Γ is the full (resp. plain) projection of G for session s, written fproj[G,s](Γ) (resp. pproj[G,s](Γ)), iff Γ = {s[p]:G[p]}p∈roles(G), where G[p] is the projection with full merging (resp. plain merging) in Def. 3.3.

**Lemma 5.9.** For all Γ, 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[G,s](Γ) ⇐⇒ live⁺(Γ).

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⁺¹] 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 OAuth2 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.

\[\Gamma_A = s[p].q &: m_1, r &: m_3, s[q].r &: m_2, p &: m_1, s[r].p &: m_3, q &: m_2\]

is consistent (hence safe), but not live nor deadlock-free: this is because its inputs/outputs, albeit dual, occur in the wrong order.

\[\Gamma_B = s[p].\mu t.q &: m_1, t, s[q].\mu t.p &: m_1, t, s[r].p &: m_2\]

is consistent, deadlock-free and safe, but not live: in fact, \(s[p], s[q]\) reduce infinitely, but \(s[r]\) cannot fire its input (violating \([L \& G]\) in Fig. 5).

\[\Gamma_C = s[p].S, s[q].p &: m(S) . end\]

with \(S = \mu t.q &: m(t) . end\) (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 \(t\) occurs as payload in \(S\), which is disallowed by Def. 3.3 and Def. 3.8. Notably, \(\Gamma_C\) types the process below (from \[Bernardi and Hennessy 2016, Ex. 1.2\]): it creates infinitely many sessions \(s'\) where \(p\) and \(q\) exchange one message \(m\) (note that this process, although deadlock-free, does not satisfy Def. 5.3(2)).

\[0 \vdash \Gamma_C \vdash def\]

\[X(x:S, y:p &: m(S)) = P\] in \(X(s[p], s[q])\)

where \(P = (\nu s' . \Gamma_C' )\) \(\langle x[q] &: m(z), y[p] \sum m(z) . X(z, s'[q])\rangle\) with \(\Gamma_C' = s'[p].S, s'[q].p &: m(S) . end\)

**Remark 5.12.** By Lemma 5.9(1,9), our general session type system instantiated with \(\varphi = fproj_{G,s}\) 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 \([T-vClassic G]\) in §3 is valid in our theory, and we can type our opening example (Ex. 4.5), 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 \((\nu s) P\) has a desired property from Def. 5.1, we can correspondingly instantiate \(\varphi\) in Def. 4.4, and check if the judgement “\(0 \vdash (\nu s) \Gamma P\) 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. 4.9, 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

\[\varphi \in \{\text{consistent, fproj}_{G,s}, \text{pproj}_{G,s}, \text{safe, term, nterm, df, live, live}^+, \text{live}^{++}\}\] (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:

$$\Gamma' = \{s[a]: s @ \text{query}(\text{Str}). s @ \text{price}(\text{Int}). \mu t. b @ \text{split}(\text{Int}). b & \{\text{yes}(\text{Int}). s @ \text{buy}, \text{no}. t\}
\{s[s]: a @ \text{query}(\text{Str}). a @ \text{price}(\text{Int}). a & \{\text{buy}. \text{end}, \text{no}. \text{end}\} \} \}$$

i.e., the types of aEalice and bOb in $\Gamma'$ are supertypes (Def. 2.5) of those in $\Gamma$: aEalice never chooses to send cancel to bOb, who in turn always answers no to all split proposals. We have $\Gamma \leq \Gamma'$ (Def. 2.5) and $\Gamma'$ is safe (Lemma 4.3), but not live: after sending the price, the store will wait for either buy or no from aEalice, but neither message will ever be sent, while aEalice and bOb loop by exchanging split/no. Consequently, a process $P$ typed by $\Gamma'$ can have two sub-processes implementing aEalice and bOb 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, $P \equiv \bigcup_{p \in I} P_p$, each $P_p$ having guarded definitions and either being $\emptyset$ (up-to $\equiv$), or only playing role $p$ in $s$. Then, (1) $d(\Gamma)$ implies that $P$ is deadlock-free; (2) $s$ 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. 4.9).

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 $[T\text{Gen-}\vee]$ (Def. 4.4), 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 $s @ p & q (S)$ for input, or $s @ p & q (S)$ 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:

$$\phi ::= T | \bot | [\alpha] \phi | (\alpha) \phi | \phi_1 \land \phi_2 | \phi_1 \lor \phi_2 | \phi_1 \Rightarrow \phi_2 | \mu Z. \phi | \nu Z. \phi | Z | V. \phi | \exists d. \phi$$

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 ($T/\bot$), i.e., accept any/no typing context; box modality $[\alpha] \phi$ ("for all transitions with label $\alpha$, the reached typing context must satisfy..."
diamond modality $\langle \alpha \rangle \phi$ ("for some transition with label $\alpha$, the reached typing context satisfies $\phi$"); implication $\Rightarrow$; least/greatest fixed point $\mu Z. \phi / v Z. \phi$, allowing to iterate $\phi$ for a finite/infinite number of times; a variable $Z$, for iteration; and universal/existential quantification $\forall d. \phi / \exists d. \phi$. 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 $\Gamma = s[r]:r'@\text{msg}(\text{Str}).\text{end}$, then we have $\Gamma \models s[r]:r'@\text{msg}(\text{Str}).\text{end}$, (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 $\Gamma' = \Gamma; s[r']:r@\text{msg}(\text{Str}).\text{end}$, then we have both $\Gamma' \models \phi$ and $\Gamma' \models \phi' — and thus, $\Gamma' \models \phi \lor \phi'$.

**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 $v Z.\ldots$ and the clause $\ldots \land [s[p;q;m]Z$;

- **deadlock-freedom (2)** checks whether communication is possible; if not ("$\forall s. \forall p. \forall q. \forall m. [s[p;q;m]] \bot$", 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 $v Z.\ldots$ and $\ldots \land \forall s. \forall p. \forall q. \forall m. [s[p;q;m]]Z$;

- **termination (3)** is similar to deadlock-freedom, but uses a $\text{least}$ fixed point $\mu Z.\ldots$: hence, the clause $\ldots \land \forall s. \forall p. \forall q. \forall m. [s[p;q;m]]Z$ can only iterate for a $\text{finite}$ 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 ($v Z.\ldots$ and $\ldots \land \forall s. \forall p. \forall q. \forall m. [s[p;q;m]]Z$), some further communication is possible ($\exists s. \exists p. \exists q. \exists m. (s[p;q;m]) \top$);

- **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 $\ldots \lor \exists s. \exists p. \exists q. \exists m. (s[p;q;m])Z'$ for a finite number of times. The top-level greatest fixed point $v Z.\ldots$ repeats the check for all contexts reachable via communication (clause $\ldots \land \forall s. \forall p. \forall q. \forall m. [s[p;q;m]]Z$);

- **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 $\text{live}^+$ 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 $\text{live}^{++}$ in Fig. 5.

**Implementation.** This paper has a companion artifact: a toolkit, called $\text{mpstk}$ ("MultiParty Session Types toolKit"), that verifies the properties listed Fig. 5 (and described in Ex. 6.2). It is available at:

https://alcestes.github.io/mpstk

Internally, $\text{mpstk}$ uses the mCRL2 model checker [Groote and Mousavi 2014], in combination with the theory in §2.2 and §4 (e.g., $\text{mpstk}$ checks subtyping, as per Def. 2.5). We used $\text{mpstk}$ to verify
the protocols in Fig. 4: the results are in Table 1. We also measured the time needed to verify each case: the results are 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 Γ, whereas the input and output transitions of Γ 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 Γ.

7 ASYNCHRONOUS MULTIPARTY SESSION π-CALCULUS

In its original formulation [Bettini et al. 2008; Honda et al. 2008], the MPST π-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. Due to space limits, we summarise the main results from [Scalas and Yoshida 2018a].

Asynchronous MPST. We give an intuition of the asynchronous calculus with an example:

\[ s[p][q] \oplus m(s'[r]).P \quad | \quad s[q][p] \sum m(x).Q \quad | \quad s \triangleright \epsilon \]

\[ \rightarrow \quad P \quad | \quad s[q][p] \sum m(x).Q \quad | \quad s \triangleright (p,q,m(s'[r])) \cdot \epsilon \quad \rightarrow \quad P \quad | \quad Q[s'[r]/x] \quad | \quad s \triangleright \epsilon \]  

(10)

In the topmost process, \( s \triangleright \epsilon \) is the (empty) message queue of session \( s \) (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:

\[ \Theta \cdot \Gamma \vdash_S P \]  

(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:

\[ \Gamma = s[p],[q]@m(S').S , s[q],[p]@m(S').T , s'[r]:S' \]

\[ \Gamma' = s[p],[q]@m(S') \cdot e , s[q],[p]@m(S').T , s'[r]:S' \]

\[ \Gamma'' = s[p],s[q]:T , s'[r]:S' \]  

(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 \( M = q[m(S')] \cdot e \) is a message queue type (abstracting the pending messages sent through \( s[p] \)), followed by the continuation type \( S \). In \( \Gamma'' \), the topmost queued message type matches the branching type of \( s[q] \): their interaction leads to \( \Gamma''' \), 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]:

If \( \Theta \cdot \Gamma \vdash_S P \) and \( \exists \Gamma_0 \) such that \( \Gamma, \Gamma_0 \) consistent and \( P \rightarrow P' \),

then \( \exists \Gamma', \Gamma''_0 \) consistent: \( \Gamma, \Gamma_0 \rightarrow^* \Gamma', \Gamma_0' \) 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 \( \varphi \), 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 \( \Theta \cdot \Gamma \vdash \Gamma \) with \( \Gamma \) asynchronously safe. Then, \( \Gamma \rightarrow \Gamma' \) implies \( \exists \Gamma'' \) asynchronously safe such that \( \Gamma \rightarrow S \Gamma' \) and \( \Theta \cdot \Gamma'' \vdash \Gamma' \).

We define asynchronous variants of \( \varphi \), similar to those in Fig.5; and by suitably instantiating \( \varphi \), we ensure that typed async processes are deadlock-free/live, similarly to Thm.5.15.

**(Un-)Decidability of Type Checking.** A result akin to Thm.4.9 holds for async MPST.

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

However, under asynchrony we do not have a decidability result for \( \varphi \) 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 \( \varphi \) through various sound approximations:

\( \Theta \cdot \Gamma \vdash \Gamma \) consistency is decidable, and implies asynchronous safety;

\( \Theta \cdot \Gamma \vdash \Gamma \) 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+;

\( \Theta \cdot \Gamma \vdash \Gamma \) 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 \( \Gamma = s[p] \cdot q \oplus m_1, q \& m_2, s[q] \cdot m_\& m_2, p \& m_1 \): 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 \( \varphi \) 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 \( \varphi \), 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.4.6, Cor.4.7) are much more general than classic MPST;

**R2** if we instantiate \( \varphi \) with projection/consistency, or any property in Fig.5, then the type checking judgement \( \"\Theta \cdot \Gamma \vdash \Gamma \) with \( \varphi \" \) is decidable. This follows from Thm.4.9 and Thm.5.13;

**R3** by suitably choosing \( \varphi \) in (R2) above, we can statically guarantee that \( \Gamma \) “inherits” \( \varphi \), and has certain desired run-time properties. This is formalised in Thm.5.15;

**R4** we can implement \( \varphi \) 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 \( \varphi \) via model checking — e.g., using mCRL2, through the paper’s companion artifact (mpsttk). 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>(b) Caires and Perez [2016]; Chen [2015]; Deniélou et al. [2012]; Deniélou and Yoshida [2012]; Toninho 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 \( \varphi \) as \( \varphi = \text{pproj}_{G \downarrow} / \varphi = \text{consistent} \); 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 \( \varphi = \text{fproj}_{G \downarrow} \), 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]: 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. Such a type system is subsumed by letting \( \varphi = \text{fproj}_{G \downarrow} \) in our Def. 4.4; 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 [2018b]: 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 [2018b] 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 [2018b], 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 \( \text{live}^* / \text{live}^{++} \) (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 [2018b];

3. the branching/selection typing rules of Scalas and Yoshida [2018b] (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 2018b, 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 [2018b] 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. 4.4, this paper largely subsumes Scalas and Yoshida [2018b]’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 $\Gamma = s[p]\cdot S, s[q]\cdot T$, our Lemma 5.9 becomes:

$$\exists G: fproj_{\alpha,s}(\Gamma) \iff \exists fproj_{\alpha,s}(\Gamma)$$

Here, the leftmost “$\iff$” is due to supertyping: e.g., if we take the global type $G = p \rightarrow q: \{m, m\}'$, it projects the typing context $\Gamma = s[p]. q \oplus \{m, m\}', s[q]. p \& \{m, m\}'$, 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 $mt.q \oplus \{m(t).end$: 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 $\Gamma_C$) 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.
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; Denièlou 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. 4.6/Cor. 4.7 holds. This corresponds to the following conjecture:

*Take any $\Gamma$. If $\forall P, P': \Gamma \vdash P$ and $P \rightarrow^* P'$ implies that $P'$ has no error, then 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 $[S-\rightarrow]$ 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


