Abstract

We develop a generalization of existing Curry-Howard interpretations of (binary) session types by relying on an extension of linear logic with features from hybrid logic, in particular modal worlds that indicate domains. These worlds govern domain migration, subject to a parametric accessibility relation familiar from the Kripke semantics of modal logic. The result is an expressive new typed process framework for domain-aware, message-passing concurrency. Its logical foundations ensure that well-typed processes enjoy session fidelity, global progress, and termination. Typing also ensures that processes only communicate with accessible domains and so respect the accessibility relation.

Remarkably, our domain-aware framework can specify scenarios in which domain information is available only at runtime; flexible accessibility relations can be cleanly defined and statically enforced. As a specific application, we introduce domain-aware multiparty session types, in which global protocols can express arbitrarily nested sub-protocols via domain migration. We develop a precise analysis of these multiparty protocols by reduction to our binary domain-aware framework: complex domain-aware protocols can be reasoned about at the right level of abstraction, ensuring also the principled transfer of key correctness properties from the binary to the multiparty setting.

1 Introduction

The goal of this paper is to show how existing Curry-Howard interpretations of session types [10, 11] can be generalized to a domain-aware setting by relying on an extension of linear logic with features from hybrid logic [40, 5]. These extended logical foundations of message-passing concurrency allow us to analyze complex domain-aware concurrent systems (including those governed by multiparty protocols) in a precise and principled manner.

Software systems typically rely on communication between heterogeneous services; at their heart, these systems rely on message-passing protocols that combine mobility, concurrency, and distribution. As distributed services are often virtualized, protocols should span diverse software and hardware domains. These domains can have multiple interpretations, such as the location where services reside, or the principals on whose behalf they act. Concurrent
behavior is then increasingly domain-aware: a partner’s potential for interaction is influenced not only by the domains it is involved in at various protocol phases (its context), but also by connectedness relations among domains. Moreover, domain architectures are rarely fully specified: to aid modularity and platform independence, system participants (e.g., developers, platform vendors, service clients) often have only partial views of actual domain structures. Despite their importance in communication correctness and trustworthiness at large, the formal status of domains within typed models of message-passing systems remains unexplored.

This paper contributes to typed approaches to the analysis of domain-aware communications, with a focus on session-based concurrency. This approach specifies the intended message-passing protocols as session types [30, 31, 24]. Different type theories for binary and multiparty (n-ary) protocols have been developed. In both cases, typed specifications can be conveniently coupled with π-calculus processes [36], in which so-called session channels connect exactly two subsystems. Communication correctness usually results from two properties: session fidelity (type preservation) and deadlock freedom (progress). The former says that well-typed processes always evolve to well-typed processes (a safety property); the latter says that well-typed processes will never get into a stuck state (a liveness property).

A key motivation for this paper is the sharp contrast between (a) the growing relevance of domain-awareness in message-passing, concurrent systems and (b) the expressiveness of existing session type frameworks, binary and multiparty, which cannot adequately specify (let alone enforce) domain-related requirements. Indeed, existing session types frameworks, including those based on Curry-Howard interpretations [10, 47, 14], capture communication behavior at a level of abstraction in which even basic domain-aware assertions (e.g., “Shipper resides in domain AmazonUS”) cannot be expressed. As an unfortunate consequence, the effectiveness of the analysis techniques derived from these frameworks is rather limited.

To better illustrate our point, consider a common distributed design pattern: a middleware agent (mw) which answers requests from clients (cl), sometimes offloading the requests to a server (serv) to better manage local resource availability. In the framework of multiparty session types [32] this protocol can be represented as the global type:

\[
\text{cl} \rightarrow \text{mw} \{ \text{request}(\text{req}) \}, \text{mw} \rightarrow \text{cl} \{ \text{reply}(\text{ans}) \}, \text{mw} \rightarrow \text{serv} \{ \text{end}_1 \}, \text{serv} \rightarrow \text{mw} \{ \text{reply}(\text{ans}) \}, \text{mw} \rightarrow \text{cl} \{ \text{reply}(\text{ans}) \}
\]

The client first sends a request to the middleware, which answers back with either a reply message containing the answer or a wait message, signaling that the server will be contacted to produce the final reply. While this multiparty protocol captures the intended communication behavior, it does not capture that protocols for the middleware and the server often involve some form of privilege escalation or specific authentication—ensuring, e.g., that the server interaction is adequately isolated from the client, or that the escalation must precede the server interactions. These requirements simply cannot be represented in existing frameworks.

Our work addresses this crucial limitation by generalizing Curry-Howard interpretations of session types by appealing to hybrid logic features. We develop a logically motivated typed process framework in which worlds from modal logics precisely and uniformly define the notion of domain in session-based concurrency. At the level of binary sessions, domains manifest themselves through point-to-point domain migration and communication. In multiparty sessions, domain migration is specified choreographically through the new construct

\[
p \text{moves } \tilde{q} \text{ to } \omega \text{ for } G_1 \cap G_2,
\]

where participant \( p \) leads a migration of participants \( \tilde{q} \) to domain \( \omega \) in order to perform protocol \( G_1 \), who then migrate back to perform protocol \( G_2 \).

Consider the global type \( \text{Offload } \equiv \text{mw} \rightarrow \text{serv} \{ \text{req}(\text{data}) \}, \text{serv} \rightarrow \text{mw} \{ \text{reply}(\text{ans}) \} \) in our previous example. Our framework allows us to refactor the global type above as:

\[
\text{cl} \rightarrow \text{mw} \{ \text{request}(\text{req}) \}, \text{mw} \rightarrow \text{cl} \{ \text{reply}(\text{ans}) \}, \text{mw} \rightarrow \text{serv} \{ \text{done}_2 \}, \text{serv} \rightarrow \text{mw} \{ \text{init} \}, \text{mw} \text{moves } \text{serv} \text{ to } \text{priv}, \text{for } \text{Offload}; \text{mw} \rightarrow \text{cl} \{ \text{reply}(\text{ans}) \}
\]
By considering a first-class multiparty domain migration primitive at the type and process levels, we can specify that the offload portion of the protocol takes place after the middleware and the server migrate to a private domain w_{priv}, as well as ensuring that only accessible domains can be interacted with. For instance, the type for the server that is mechanically projected from the protocol above ensures that the server first migrates to the private domain, communicates with the middleware, and then migrates back to its initial domain.

Perhaps surprisingly, our domain-aware multiparty sessions are studied within a context of logical binary domain-aware sessions, arising from a propositions-as-types interpretation of hybrid linear logic [22, 18], with strong static correctness guarantees derived from the logical nature of the system. Multiparty domain-awareness arises through an interpretation of multiparty protocols as medium processes [7] that orchestrate the multiparty interaction while enforcing the necessary domain-level constraints and migration steps.

Contributions The key contributions of this work are:

1. A process model with explicit domain-based migration (§2). We present a session π-calculus with domains that can be communicated via novel domain movement prefixes.
2. A session type discipline for domain-aware interacting processes (§3). Building upon an extension of linear logic with features from hybrid logic [22, 18] we generalize the Curry-Howard interpretation of session types [10, 11] by interpreting (modal) worlds as domains where session behavior resides. In our system, types can specify domain migration and communication; domain mobility is governed by a parametric accessibility relation. Judgments stipulate the services used and realized by processes and the domains where sessions should be present. Our type discipline statically enforces session fidelity, global progress and, notably, that communication can only happen between accessible domains.
3. As a specific application, we introduce a framework of domain-aware multiparty sessions (§4) that uniformly extends the standard multiparty session framework of [32] with domain-aware migration and communication primitives. Our development leverages our logically motivated domain-aware binary sessions (§3) to give a precise semantics to multiparty sessions through a (typed) medium process that acts as an orchestrator of domain-aware multiparty interactions, lifting the strong correctness properties of typed processes to the multiparty setting. We show that mediums soundly and completely encode the local behaviors of participants in a domain-aware multiparty session.

We conclude with a discussion of related work (§5) and concluding remarks (§6).

2 Process Model

We introduce a synchronous π-calculus [42] with labeled choice and explicit domain migration and communication. We write ω, ω', ω'' to stand for a concrete domain (w, w', . . .) or a domain variable (α, α', . . .). Domains are handled at a high-level of abstraction, with their identities being attached to session channels. Just as the π-calculus allows for communication over names and name mobility, our model also allows for domain communication and mobility. These features are justified with the typing discipline of §3.

Definition 2.1. Given infinite, disjoint sets Λ of names (x, y, z, u, v), L of labels l_1, l_2, . . ., W of domain tags (w, w', w'') and V of domain variables (α, β, γ), respectively, the set of processes (P, Q, R) is defined by

\[
P ::= 0 \mid P \mid Q \mid x (\nu y) P \mid x \{k : P\} \mid x \alpha \mid x (y@\omega).P \mid x (y@\omega).P \mid x(\omega).P \mid x(\alpha).P
\]

CONCUR 2019
Domain-aware prefixes are present only in the last line. As we make precise in the typed setting of §3, these constructs realize mobility and domain communication, in the usual sense of the π-calculus: migration to a domain is always associated to mobility with a fresh name.

The operators \( \text{inaction} \), \( P \parallel Q \) (parallel composition) and \( (\nu y)P \) (name restriction) are standard. We then have \( x(y).P \) (send \( y \) on \( x \) and proceed as \( P \)), \( x(y).P \) (receive \( z \) on \( x \) and proceed as \( P \) with parameter \( y \) replaced by \( z \)), and \( !x(y).P \) which denotes replicated (persistent) input. The forwarding construct \( [x \leftrightarrow y] \) equates \( x \) and \( y \); it is a primitive representation of a copycat process. The last two constructs in the second line define a labeled choice mechanism: \( x \triangleright \{ l_i : P_i \}_{i \in I} \) is a process that awaits some label \( l_j \) (with \( j \in I \)) and proceeds as \( P_j \). Dually, the process \( x \triangleleft l_i \) emits a label \( l_i \) and proceeds as \( P \).

The first two operators in the third line define explicit domain migration: given a domain \( \omega \), \( x(y@\omega).P \) denotes a process that is prepared to migrate the communication actions in \( P \) on endpoint \( x \), to session \( y \) on \( \omega \). Complementarily, process \( x(y@\omega).P \) signals an endpoint \( x \) to move to \( \omega \), providing \( P \) with the appropriate session endpoint that is then bound to \( y \). In a typed setting, domain movement will be always associated with a fresh session channel.

Alternatively, this form of coordinated migration can be read as an explicit form of agreement (or authentication) in trusted domains. Finally, the last two operators in the third line define input and output of domains, \( x(\omega).P \) and \( x(\alpha).P \), respectively. These constructs allow for domain information to be obtained and propagated across processes dynamically.

Following [41], we abbreviate \( (\nu y)x(y) \) and \( (\nu y)x(y@\omega) \) as \( \bar{x}(y) \) and \( \bar{x}(y@\omega) \), respectively. In \( (\nu y)P \), \( x(y).P \), and \( x(y@\omega).P \) the distinguished occurrence of name \( y \) is binding with scope \( P \). Similarly for \( \alpha \) in \( x(\alpha).P \). We identify processes up to consistent renaming of bound names and variables, writing \( \equiv_{\alpha} \) for this congruence. \( P[x/y] \) denotes the capture-avoiding substitution of \( x \) for \( y \) in \( P \). While \textit{structural congruence} \( \equiv \) expresses standard identities on the basic structure of processes (cf. [9]), \textit{reduction} expresses their behavior.

\textit{Reduction} \( (P \rightarrow Q) \) is the binary relation defined by the rules below and closed under structural congruence; it specifies the computations that a process performs on its own.

\[
\begin{align*}
  x(y).Q \parallel x(z).P & \rightarrow Q \parallel P[y/z] \\
x(y@\omega).P \parallel x(z@\omega).Q & \rightarrow P \parallel Q[y/z] \\
(\nu x)[(x \leftrightarrow y) \parallel P] & \rightarrow (\nu y)[x(y)]P \\
P & \rightarrow Q \Rightarrow (\nu y)P \rightarrow (\nu y)Q
\end{align*}
\]

For the sake of generality, reduction allows dual endpoints with the same name to interact, independently of the domains of their subjects. The type system introduced next will ensure, among other things, \textit{local reductions}, disallowing synchronisations among distinct domains.

### 3 Domain-aware Session Types via Hybrid Logic

This section develops a new domain-aware formulation of binary session types. Our system is based on a Curry-Howard interpretation of a linear variant of so-called \textit{hybrid logic}, and can be seen as an extension of the interpretation of [10, 11] to hybrid (linear) logic. Hybrid logic is often used as an umbrella term for a class of logics that extend the expressiveness of propositional logic by considering modal \textit{worlds} as syntactic objects that occur in propositions.

As in [10, 11], propositions are interpreted as session types of communication channels, proofs as typing derivations, and proof reduction as process communication. As main novelties, here we interpret: logical worlds as \textit{domains}; the hybrid connective \( @_\omega \alpha.A \) as the type of a session that \textit{migrates} to an accessible domain \( \omega \); and type-level quantification over worlds \( \forall \alpha.A \) and \( \exists \alpha.A \) as \textit{domain communication}. We also consider a type-level operator
\[ \downarrow \alpha.A \] (read “here”) which binds the current domain of the session to \( \alpha \) in \( A \). The syntax of domain-aware session types is given in Def. 3.1, where \( w, w_1, \ldots \) stand for domains drawn from \( W \), and where \( \alpha, \beta \) and \( \omega, \omega' \) are used as in the syntax of processes.

**Definition 3.1 (Domain-aware Session Types).** The syntax of types \((A, B, C)\) is defined by

\[
A ::= 1 \mid A \rightarrow B \mid A \otimes B \mid \& \{ k_i : A_i \}_{i \in I} \mid \oplus \{ k_i : A_i \}_{i \in I} \mid !A \mid \oplus \alpha . A \mid \forall \alpha . A \mid \exists \alpha . A \mid \downarrow \alpha . A
\]

Types are the propositions of intuitionistic linear logic where the additives \( A \& B \) and \( A \oplus B \) are generalized to a labelled \( n \)-ary variant. Propositions take the standard interpretation as session types, extended with hybrid logic operators [5], with worlds interpreted as domains that are explicitly subject to an accessibility relation (in the style of [43]) that is tracked by environment \( \Omega \). Intuitively, \( \Omega \) is made up of direct accessibility hypotheses of the form \( \omega_1 \prec \omega_2 \), meaning that domain \( \omega_2 \) is accessible from \( \omega_1 \).

Types are assigned to channel names; a type assignment \( \omega : A[\omega] \) enforces the use of name \( \omega \) according to session \( A \), in the domain \( \omega \). A type environment is a collection of type assignments. Besides the accessibility environment \( \Omega \) just mentioned, our typing judgments consider two kinds of type environments: a linear part \( \Delta \) and an unrestricted part \( \Gamma \). They are subject to different structural properties: weakening and contraction principles hold for \( \Gamma \) but not for \( \Delta \). Empty environments are written as ‘\( \cdot \)’.

We then consider two judgments:

(i) \( \Omega \vdash \omega_1 \prec \omega_2 \) and (ii) \( \Omega; \Gamma; \Delta \vdash P :: z : A[\omega] \)

Judgment (i) states that \( \omega_1 \) can directly access \( \omega_2 \) under the hypotheses in \( \Omega \). We write \( \prec \) for the reflexive, transitive closure of \( \prec \), and \( \omega_1 \nprec \omega_2 \) when \( \omega_1 \nprec \omega_2 \) does not hold.

Judgment (ii) states that process \( P \) offers the session behavior specified by type \( A \) on channel \( z \); the session \( s \) resides at domain \( \omega \), under the accessibility hypotheses \( \Omega \), using unrestricted sessions in \( \Gamma \) and linear sessions in \( \Delta \). Note that each hypothesis in \( \Gamma \) and \( \Delta \) is labeled with a specific domain. We omit \( \Omega \) when it is clear from context.

**Typing Rules** Selected typing rules are given in Fig. 1; see [9] for the full listing. Right rules (marked with R) specify how to offer a session of a given type, left rules (marked with L) define how to use a session. The hybrid nature of the system induces a notion of well-formedness of sequents: a sequent \( \Omega; \Gamma; \Delta \vdash P :: z : C[\omega] \) is well-formed if \( \Omega \vdash \omega_1 \nprec \omega_2 \) for every \( x : A[\omega_2] \in \Delta \), which we abbreviate as \( \Omega \vdash \omega_1 \prec \omega \Delta \), meaning that all domains mentioned in \( \Delta \) are accessible from \( \omega_1 \) (not necessarily in a single direct step). No such domain requirement is imposed on \( \Gamma \). If an end sequent is well-formed, every sequent in its proof will also be well-formed. All rules (read bottom-up) preserve this invariant; only (cut), (copy), (\( \otimes \)R), (\( \forall \)L) and (\( \exists \)R) require explicit checks, which we discuss below. This invariant statically excludes interaction between sessions in accessible domains (cf. Theorem 3.7).

We briefly discuss some of the typing rules, first noting that we consider processes modulo structural congruence; hence, typability is closed under \( \equiv \) by definition. Type \( A \rightarrow B \) denotes a session that inputs a session of type \( A \) and proceeds as \( B \). To offer \( z : A \rightarrow B \) at domain \( \omega \), we input \( y \) along \( z \) that will offer \( A \) at \( \omega \) and proceed, now offering \( z : B \) at \( \omega \):

\[
(\rightarrow R) \quad \Omega; \Gamma; \Delta \vdash z : A[\omega] \quad \Omega; \Gamma; \Delta \vdash (y).P :: z : A \rightarrow B[\omega] \quad (\otimes R) \quad \Omega; \Gamma; \Delta_1 \vdash P :: y : A[\omega] \quad \Omega; \Gamma; \Delta_2 \vdash Q :: z : B[\omega] \quad \Omega; \Gamma; \Delta_1, \Delta_2 \vdash \tau(y).(P | Q) :: z : A \otimes B[\omega]
\]

Dually, \( A \otimes B \) denotes a session that outputs a session that will offer \( A \) and continue as \( B \).

To offer \( z : A \otimes B \), we output a fresh name \( y \) with type \( A \) along \( z \) and proceed offering \( z : B \).
The (cut) rule allows us to compose process $P$, which offers $x:A[\omega_2]$, with process $Q$, which uses $x:A[\omega_2]$ to offer $z:C[\omega_1]$. We require that domain $\omega_2$ is accessible from $\omega_1$ (i.e., $\omega_1 \prec^* \omega_2$). We also require $\omega_1 \prec^* \Delta_1$: the domains mentioned in $\Delta_1$ (the context for $P$) must be accessible from $\omega_1$, which follows from the transitive closure of the accessibility relation ($\prec^*$) using the intermediary domain $\omega_2$. As in [10, 11], composition binds the name $x$:

$$\Omega \vdash \omega_1 \prec^* \omega_2 \quad \Omega \vdash \omega_1 \prec^* \Delta_1 \quad \Omega; \Gamma; \Delta_1 \vdash P : x:A[\omega_2] \quad \Omega; \Gamma; \Delta_2, x:A[\omega_2] \vdash Q : z:C[\omega_1]$$

Hence, a process offering a behavior $z:\@_{\omega_2}A$ at $\omega_1$ ensures: (i) behavior $A$ is available at $\omega_2$ along a fresh session channel $y$ that is emitted along $z$ and (ii) $\omega_2$ is directly accessible from $\omega_1$. To maintain well-formedness of the sequent we also must check that all domains in $\Delta$ are still accessible from $\omega_2$. Dually, using a service $x:\@_{\omega_2}A[\omega_2]$ entails receiving a channel $y$ that will offer behavior $A$ at domain $\omega_3$ (and also allowing the usage of the fact that $\omega_2 \prec \omega_3$).

Domain-quantified sessions introduce domains as fresh parameters to types: a particular service can be specified with the ability to refer to any existing directly accessible domain (via universal quantification) or to some a priori unspecified accessible domain:

$$\Omega, \omega_1 \prec \alpha; \Gamma; \Delta \vdash P : x:A[\omega_1] \quad \alpha \not\in \Omega, \Gamma, \Delta, \omega_1$$

Rule (\forall R) states that a process seeking to offer $\forall \alpha.A[\omega_1]$ denotes a service that is located at domain $\omega_1$ but that may refer to any fresh domain directly accessible from $\omega_1$ in its specification (e.g. through the use of @). Operationally, this means that the process must be ready to receive from its client a reference to the domain being referred to in the type, which is bound to $\alpha$ (occurring fresh in the typing derivation). Dually, Rule (\forall L) indicates that a
process interacting with a service of type $x:\forall \alpha.A[\omega_2]$ must make concrete the domain that
is directly accessible from $\omega_2$ it wishes to use, which is achieved by the appropriate output
action. Rules ($\exists L$) and ($\exists R$) for the existential quantifier have a dual reading.

Finally, the type-level operator $\downarrow \alpha.A$ allows for a type to refer to its current domain:

$$\begin{align*}
\downarrow L : \Omega; \Gamma; \Delta \vdash P :: z: A[\omega/\alpha][\omega] & \\
\downarrow R : \Omega; \Gamma; \Delta \vdash P :: z: \forall \alpha.A[\omega] &
\end{align*}$$

The typing rules that govern $\downarrow \alpha.A$ are completely symmetric and produce no action at the
process level, merely instantiating the domain variable $\alpha$ with the current domain $\omega$ of the
session. As will be made clear in §4, this connective plays a crucial role in ensuring the
correctness of our analysis of multiparty domain-aware sessions in our logical setting.

By developing our type theory with an explicit domain accessibility judgment, we can
consider the accessibility relation as a parameter of the framework. This allows changing
accessibility relations and their properties without having to alter the entire system. To
consider the simplest possible accessibility relation, the only defining rule for accessibility
would be Rule (whyh) in Fig. 1. To consider an accessibility relation which is an equivalence
relation we would add reflexivity, transitivity, and symmetry rules to the judgment.

Discussion and Examples  Being an interpretation of hybridized linear logic, our domain-
aware theory is conservative wrt the Curry-Howard interpretation of session types in [10, 11],
in the following sense: the system in [10, 11] corresponds to the case where every session
resides at the same domain. As in [10, 11], the sequent calculus for the underlying (hybrid)
linear logic can be recovered from our typing rules by erasing processes and name assignments.

Conversely, a fundamental consequence of our hybrid interpretation is that it refines the
session type structure in non-trivial ways. By requiring that communication only occurs
between sessions located at the same (or accessible) domain we effectively introduce a new
layer of reasoning to session type systems. To illustrate this feature, consider the following
session type WStore, which specifies a simple interaction between a web store and its clients:

$$WStore \triangleq \text{addCart} \rightarrow &\{\text{buy} : \text{Pay}, \text{quit} : 1\} \quad \text{Pay} \triangleq \text{CCNum} \rightarrow \oplus\{\text{ok} : \text{Rcpt} \odot 1, \text{nok} : 1\}$$

WStore allows clients to checkout their shopping carts by emitting a buy message or to quit.
In the former case, the client pays for the purchase by sending their credit card data. If
a banking service (not shown) approves the transaction (via an ok message), a receipt is
emitted. Representable in existing session type systems (e.g. [10, 47, 31]), types WStore and
Pay describe the intended communications but fail to capture the crucial fact that in practice
the client’s sensitive information should only be requested after entering a secure domain. To
address this limitation, we can use type-level domain migration to refine WStore and Pay:

$$\begin{align*}
WStore_{sec} & \triangleq \text{addCart} \rightarrow &\{\text{buy} : \text{@sec Pay}_{\text{bnk}}, \text{quit} : 1\} \\
\text{Pay}_{\text{bnk}} & \triangleq \text{CCNum} \rightarrow \oplus\{\text{ok} : (@_{\text{bnk}}\text{Rcpt}) \odot 1, \text{nok} : 1\}
\end{align*}$$

WStore_{sec} decrees that the interactions pertinent to type Pay_{bnk} should be preceded by a
migration step to the trusted domain sec, which should be directly accessible from WStore_{sec}’s
current domain. The type also specifies that the receipt must originate from a bank domain
bnk (e.g., ensuring that the receipt is never produced by the store without entering bnk).
When considering the interactions with a client (at domain c) that checks out their cart, we
reach a state that is typed with the following judgment:

$$c \prec ws; :: x:\forall \text{@sec Pay}_{\text{bnk}}[ws] \vdash \text{Client} :: z:\forall \text{@sec 1}[c]$$
At this point, it is impossible for a (typed) client to interact with the behavior that is protected by the domain \( \text{sec} \), since it is not the case that \( c \prec^* \text{sec} \). That is, no judgment of the form \( c \prec \text{ws} ; x \cdot \text{Pay}_{\text{bank}} [\text{sec}] \mid \text{Client}' :: z : T [c] \) is derivable. This ensures, e.g., that a client cannot exploit the payment platform of the web store by accessing the trusted domain in unforeseen ways. The client can only communicate in the secure domain after the web store service has migrated accordingly, as shown by the judgment

\[
c \prec \text{ws} ; \text{ws} \prec \text{sec} ; x \cdot z \cdot \text{Pay}_{\text{bank}} [\text{sec}] \mid \text{Client}' :: z' : 1 [\text{sec}].
\]

**Technical Results** We state the main results of type safety via type preservation (Theorem 3.3) and global progress (Theorem 3.4). These results directly ensure session fidelity and deadlock-freedom. Typing also ensures termination, i.e., processes do not exhibit infinite reduction paths (Theorem 3.5). We note that in the presence of termination, our progress result ensures that communication actions are always guaranteed to take place. Moreover, as a property specific to domain-aware processes, we show domain preservation, i.e., processes respect their domain accessibility conditions (Theorem 3.7). The formal development of these results relies on a domain-aware labeled transition system [9], defined as a simple generalization of the early labelled transition system for the session π-calculus given in [10, 11].

**Type Safety and Termination.** Following [10, 11], our proof of type preservation relies on a simulation between reductions in the session-typed π-calculus and logical proof reductions.

**Lemma 3.2 (Domain Substitution).** Suppose \( \Omega \vdash \omega_1 \prec \omega_2 \). Then we have:

- If \( \Omega, \omega_1 \prec \alpha; \Omega'; \Gamma; \Delta \vdash P :: z: A[\omega] \) then 
  \( \Omega, \Omega' \omega_2/\alpha, \Gamma; \Delta \omega_2/\alpha \vdash P \omega_2/\alpha :: z: A[\omega_2/\alpha] \).

- If \( \Omega, \omega_2 \prec \alpha, \Omega'; \Gamma; \Delta \vdash P :: z: A[\omega] \) then 
  \( \Omega, \Omega' \omega_1/\alpha, \Gamma; \Delta \omega_1/\alpha \vdash P \omega_1/\alpha :: z: A[\omega_1/\alpha] \).

Safe domain communication relies on domain substitution preserving typing (Lemma 3.2).

**Theorem 3.3 (Type Preservation).** If \( \Omega; \Gamma; \Delta \vdash P :: z: A[\omega] \) and \( P \rightarrow Q \) then 
\( \Omega; \Gamma; \Delta \vdash Q :: z: A[\omega] \).

![Figure 1 Typing Rules (Excerpt – see [9])](image-url)
Proof (Sketch). The proof mirrors those of [10, 11, 8, 44], relying on a series of lemmas relating the result of dual process actions (via our LTS semantics) with typable parallel
compositions through the (cut) rule [9]. For session type constructors of [10], the results are unchanged. For the domain-aware session type constructors, the development is identical
that of [8] and [44], which deal with communication of types and data terms, respectively.

Following [10, 11], the proof of global progress relies on a notion of a live process, which intuitively consists of a process that has not yet fully carried out its ascribed session behavior, and thus is a parallel composition of processes where at least one is a non-replicated process, guarded by some action. Formally, we define live(P) if and only if P ≡ (νn)(π.Q | R), for some R, names n, and a non-replicated guarded process π.Q.

△ Theorem 3.4 (Global Progress). If Ω; Γ; ∆ ⊢ P : x::A[ω] and live(P) then ∃Q s.t. P ⇓ Q.

Note that Theorem 3.4 is without loss of generality since using the cut rules we can compose arbitrary well-typed processes together and x need not occur in P due to Rule (1R).

Termination (strong normalization) is a relevant property for interactive systems: while from a global perspective they are meant to run forever, at a local level participants should always react within a finite amount of time, and never engage into infinite internal behavior.

We say that a process P terminates, noted P ⇓, if there is no infinite reduction path from P.

△ Theorem 3.5 (Termination). If Ω; Γ; ∆ ⊢ P : x::A[ω] then P ⇓.

Proof (Sketch). By adapting the linear logical relations given in [38, 39, 8]. For the system in §3 without quantifiers, the logical relations correspond to those in [38, 39], extended to carry over Ω. When considering quantifiers, the logical relations resemble those proposed for polymorphic session types in [8], noting that no impredicativity concerns are involved.

Domain Preservation. As a consequence of the hybrid nature of our system, well-typed processes are guaranteed not only to faithfully perform their prescribed behavior in a deadlock-free manner, but they also do so without breaking the constraints put in place on domain accessibility given by our well-formedness constraint on sequents.

△ Theorem 3.6. Let E be a derivation of Ω; Γ; ∆ ⊢ P : z::A[ω]. If Ω; Γ; ∆ ⊢ P : z::A[ω] is well-formed then every sub-derivation in E well-formed.

While inaccessible domains can appear in Γ, such channels can never be used and thus can not appear in a well-typed process due to the restriction on the (copy) rule. Combining Theorems 3.3 and 3.6 we can then show that even if a session in the environment changes domains, typing ensures that such a domain will be (transitively) accessible:

△ Theorem 3.7. Let (1) Ω; Γ; ∆ ⊳ (νx)(P | Q) : z::A[ω], (2) Ω; Γ; ∆ ⊢ P : x::B[ω'], and (3) Ω; Γ; ∆', x::B[ω'] ⊢ Q : z::A[ω]. If (νx)(P | Q) → (νx)(P' | Q') then: (a) Ω; Γ; ∆ ⊳ P' : x'.B'[ω''], for some x', B', ω''; (b) Ω; Γ; ∆'; x'.B'[ω''] ⊳ Q' : z::A[ω]; (c) ω ≺∗ ω''.

4 Domain-Aware Multiparty Session Types

We now shift our attention to multiparty session types [32]. We consider the standard ingredients: global types, local types, and the projection function that connects the two. Our global types include a new domain-aware construct, p moves q.to.ω for G1; G2; our local types exploit the hybrid session types from Def. 3.1. Rather than defining a separate type system based on local types for the process model of §2, our analysis of multiparty protocols extends
the approach defined in [7], which uses medium processes to characterize correct multiparty implementations. The advantages are twofold: on the one hand, medium processes provide a precise semantics for global types; on the other hand, they enable the principled transfer of the correctness properties established in §3 for binary sessions (type preservation, global progress, termination, domain preservation) to the multiparty setting. Below, participants are ranged over by p, q, r, ...; we write ∪q to denote a finite set of participants q1, ..., qn.

Besides the new domain-aware global type, our syntax of global types includes constructs from [32, 21]. We consider value passing in branching (cf. U below), fully supporting delegation. To streamline the presentation, we consider global types without recursion.

**Definition 4.1 (Global and Local Types).** Define global types (G) and local types (T) as

\[
U ::= \text{bool | nat | str | ... | } T
\]

\[
G ::= \text{end | } p \rightarrow q \{ k(U_i), G_i \} \subseteq I \mid \text{p moves } \tilde{q} \text{ to } \omega \text{ for } G_1 ; G_2
\]

\[
T ::= \text{end | } \{ k(U_i), T_i \} \subseteq I \mid \text{p moves } \tilde{q} \text{ to } \omega \text{ for } G_1 ; G_2
\]

The completed global type is denoted end. Given a finite I and pairwise different labels, p ↠ q \{ k(U_i), G_i \} \subseteq I specifies that by choosing label k_i, participant p may send a message of type U_i to participant q, and then continue as G_i. We decree p ≠ q, so reflexive interactions are disallowed. The global type \text{p moves } \tilde{q} \text{ to } \omega \text{ for } G_1 ; G_2 specifies the migration of participants p, \tilde{q} to domain ω in order to perform the sub-protocol G_1; this migration is lead by p. Subsequently, all of p, \tilde{q} migrate from ω back to their original domains and protocol G_2 is executed. This intuition will be made precise by the medium processes for global types (cf. Def. 4.8). Notice that G_1 and G_2 may involve different sets of participants. In writing \text{p moves } \tilde{q} \text{ to } \omega \text{ for } G_1 ; G_2 we assume two natural conditions: (a) all migrating participants intervene in the sub-protocol (i.e., the set of participants of G_1 is exactly p, \tilde{q}) and (b) domain ω is accessible (via ∼) by all these migrating participants in G_1. While subprotocols and session delegation may appear as similar, delegation supports a different idiom altogether, and has no support for domain awareness. Unlike delegation, with subprotocols we can specify a point where some of the participants perform a certain protocol within the same multiparty session and then return to the main session as an ensemble.

**Definition 4.2.** The set of participants of G (denoted part(G)) is defined as: part(end) = ∅, part(p ↠ q \{ k(U_i), G_i \} \subseteq I) = \{ p, q \} ∪ \bigcup_{i \in I} part(G_i), part(p \text{ moves } \tilde{q} \text{ to } \omega \text{ for } G_1 ; G_2) = \{ p \} ∪ \tilde{q} ∪ part(G_1) ∪ part(G_2). We sometimes write p ∈ G to mean p ∈ part(G).

Global types are projected onto participants so as to obtain local types. The terminated local type is end. The local type \text{p moves } \tilde{q} \text{ to } \omega \text{ for } G_1 ; G_2 is the image of (\forall I) part(G). We now define (merge-based) projection for global types [21]. To this end, we rely on a merge operator on local types, which in our case considers messages U.

**Definition 4.3 (Merge).** We define ⊔ as the commutative partial operator on base and local types such that bool ⊔ bool = bool (and analogously for other base types), and

1. \( T \sqcup T = T \), where T is one of the following: end, \{ k(U_i), T_i \} \subseteq I, \forall \alpha T, or \exists \alpha T;

2. \( \{ k(U_k), T_k \} \subseteq K, k \sqcup \{ l(U_j'), T_j' \} \subseteq J \rightarrow\)

\( \{ (k(U_k), T_k) \subseteq K \setminus J \cup \{ l(U_j') \} \subseteq J \}\) \subseteq K ∪ \{ k(U_k ⊔ U_l'), (T_k ⊔ T_j') \} \subseteq K ∪ J)

and is undefined otherwise.
Therefore, for \( U_1 \cup U_2 \) to be defined there are two options: (a) \( U_1 \) and \( U_2 \) are identical base, terminated, selection, or “hybrid” local types; (b) \( U_1 \) and \( U_2 \) are branching types, but not necessarily identical: they may offer different options but with the condition that the behavior in labels occurring in both \( U_1 \) and \( U_2 \) must be mergeable.

To define projection and medium processes for the global type \( p \text{ moves } q \) to \( \omega \) for \( G_1 \); \( G_2 \), we require ways of “fusing” local types and processes. The intent is to capture in a single (sequential) specification the behavior of two distinct (sequential) specifications, i.e., those corresponding to protocols \( G_1 \) and \( G_2 \). For local types, we have the following definition, which safely appends a local type to another:

**Definition 4.4 (Local Type Fusion).** The fusion of \( T_1 \) and \( T_2 \), written \( T_1 \circ T_2 \), is given by:

\[
\begin{align*}
p[l(U_i),T_i]_{i \in I} \circ T & = p[l(U_i),(T_i \circ T)]_{i \in I} \quad \text{end } \circ T = T \\
p[l(U_i),T_i]_{i \in I} \circ T & = p[l(U_i),(T_i \circ T)]_{i \in I} \quad (\exists \alpha.T_1) \circ T = \exists \alpha.(T_1 \circ T) \\
(\forall a.T) \circ T & = \forall a.(T_1 \circ T) \quad (@_a.T_1) \circ T = @_a.(T_1 \circ T)
\end{align*}
\]

This way, e.g., if \( T_1 = \exists \alpha.@_\alpha.p[l\{\text{Int}\}.\text{end},l\{\text{Bool}\}.\text{end}] \) and \( T_2 = @_\omega.q[l\{\text{Str}\}.\text{end}] \), then \( T_1 \circ T_2 = \exists \alpha.@_\alpha.p[l\{\text{Int}\},@_\omega.q[l\{\text{Str}\}.\text{end}],l\{\text{Bool}\},@_\omega.q[l\{\text{Str}\}.\text{end}] \). We can now define:

**Definition 4.5 (Merge-based Projection [21]).** Let \( G \) be a global type. The merge-based projection of \( G \) under participant \( r \), denoted \( G[r] \), is defined as \( \text{end} \mid r = \text{end} \) and

\[
\begin{align*}
p & \to q[l(U_i),G_i]_{i \in I} \quad r = p \\
p[l(U_i),G_i]_{i \in I} & \quad r = q \\
\cup_{i \in I} G_i[r] & \quad \text{otherwise (\& as in Def. 4.3)}
\end{align*}
\]

When no side condition holds, the map is undefined.

The projection for the type \( p \text{ moves } q \) to \( \omega \) for \( G_1 \); \( G_2 \) is one of the key points in our analysis.

The local type for \( p \), the leader of the migration, starts by binding the identity of its current domain (say, \( \omega_q \)) to \( \beta \). Then, the (fresh) domain \( \omega \) is communicated, and there is a migration step to \( \omega \), which is where protocol \( G_1[p] \) will be performed. Finally, there is a migration step from \( \omega \) back to \( \omega_p \); once there, the protocol \( G_2[p] \) will be performed. The local type for all of \( q, q \in q \) follows accordingly: they expect \( \omega \) from \( p \); the migration from their original domains to \( \omega \) (and back) is as for \( p \). For participants in \( G_1 \), the fusion on local types (Def. 4.4) defines a local type that includes the actions for \( G_1 \) but also for \( G_2 \), if any: a participant in \( G_1 \) need not be involved in \( G_2 \). Interestingly, the resulting local types \( \downarrow \beta.(@_a.@_\alpha G_1[p] \circ @_\beta G_2[r] \quad \text{if } r = p \\
\downarrow \beta.(@_a.@_\alpha G_1[q] \circ @_\beta G_2[q] \\
\text{otherwise}
\]

The following notion of well-formedness for global types is standard:

**Definition 4.6 (Well-Formed Global Types [32]).** We say that global type \( G \) is well-formed (WF, in the following) if the projection \( G[r] \) is defined for all \( r \in G \).

**Analyzing Global Types via Medium Processes.** A medium process is a well-typed process from §2 that captures the communication behavior of the domain-aware global types of Def. 4.1. Here we define medium processes and establish two fundamental characterization results for them (Theorems 4.11 and 4.12). We shall consider names indexed by participants: given a name \( c \) and a participant \( p \), we use \( c_p \) to denote the name along which the session behavior of \( p \) will be made available. This way, if \( p \neq q \) then \( c_p \neq c_q \). To define mediums, we need to append or fuse sequential processes, just as Def. 4.4 fuses local types:
Definition 4.7 (Fusion of Processes). We define $\circ$ as the partial operator on well-typed processes such that (with $\pi \in \{c(y), c(\omega), c(\alpha), c(y \otimes \omega), c(y \otimes \omega), c(\text{true})\}$):

$$c(y).\langle [u \leftrightarrow v] | \pi \rangle \begin{cases} \text{undefined} & \text{if } \pi \circ Q \neq Q \\
\circ Q & \text{if } \pi \circ Q = Q \\
\circ (\pi \circ Q) & \text{if } \pi \circ Q \neq \pi \circ Q \\
\circ Q & \text{otherwise.}
\end{cases}$$

and is undefined otherwise.

The previous definition suffices to define a medium process (or simply medium), which uses indexed names to uniformly capture the behavior of a global type:

Definition 4.8 (Medium Process). Let $G$ be a global type (cf. Def. 4.1), $\bar{c}$ be a set of indexed names, and $\bar{\omega}$ a set of domains. The medium of $G$, denoted $\text{Med}[G](\bar{c})$, is defined as:

$$0 \begin{cases} \text{if } G = \text{end} \\
\circ \{l : cp(u).cq \triangleq \langle [u \leftrightarrow v] | \text{Med}[G_i](\bar{c}) \rangle \}_{i \in I} & \text{if } G = p \rightarrow q \{l(U_i).G_i\}_{i \in I} \\
0 & \text{if } G = p \text{ moves } q_1, \ldots, q_n \text{ to } w \text{ for } G_1 : G_2 \\
\circ \{l : cp(u).cq \triangleq \langle [u \leftrightarrow v] | \text{Med}[G_1](\bar{c}) \rangle \circ \text{Med}[G_2](\bar{c}) \} & \text{as in Def. 4.7.}
\end{cases}$$

The medium for $G = p \rightarrow q \{l(U_i).G_i\}_{i \in I}$ exploits four prefixes to mediate in the interaction between the implementations of $p$ and $q$: the first two prefixes (on name $cp$) capture the label selected by $p$ and the subsequently received value; the third and fourth prefixes (on name $cq$) propagate the choice and forward the value sent by $p$ to $q$. We omit the forwarding and value exchange when the interaction does not involve a value payload.

The medium for $G = p \text{ moves } q_1, \ldots, q_n \text{ to } w \text{ for } G_1 : G_2$ showcases the expressivity and convenience of our domain-aware process framework. In this case, the medium’s behavior takes place through the following steps: First, $\text{Med}[G](\bar{c})$ inputs a domain identifier (say, $\omega$) from $p$ which is forwarded to $q_1, \ldots, q_n$, the other participants of $G_1$. Secondly, the roles $p, q_1, \ldots, q_n$ migrate from their domains $\omega_p, \omega_{q_1}, \ldots, \omega_{q_n}$ to $\omega$. At this point, the medium for $G_1$ can execute, keeping track the current domain $\omega$ for all participants. Finally, the participants of $G_1$ migrate back to their original domains and the medium for $G_2$ executes.

Recalling the domain-aware global type of § 1, we produce its medium process:

$$\begin{array}{l}
\begin{cases}
\text{request} : c_{\text{cl}}(r).c_{\text{sw}} < \text{request}; \text{sw}(v).\langle [r \leftrightarrow v] | c_{\text{sw}} \rangle & \text{if } G = \text{end} \\
\text{reply} : c_{\text{sw}}(a).c_{\text{cl}} < \text{reply}; \text{cl}(n).\langle [a \leftrightarrow n] | c_{\text{sw}} \rangle & \text{if } G = p \rightarrow q \{l(U_i).G_i\}_{i \in I} \\
\text{init} : c_{\text{sw}} < \text{init}; c_{\text{sw}}(u_{\text{priv}}) & \text{if } G = p \text{ moves } q_1, \ldots, q_n \text{ to } w \text{ for } G_1 : G_2 \\
\text{vsw}(y_{\text{sw}} \otimes u_{\text{sw}}), c_{\text{sw}}(y_{\text{sw}} \otimes u_{\text{sw}}), \text{Vsw}^\text{init}(y_{\text{sw}} \otimes u_{\text{sw}}), \text{Vsw}^\text{offload}(y_{\text{sw}} \otimes u_{\text{sw}}), \text{Vsw}^\text{vsw}(y_{\text{sw}} \otimes u_{\text{sw}}), c_{\text{sw}}(y_{\text{sw}} \otimes u_{\text{sw}}) & \text{as in Def. 4.7.}
\end{cases}
\end{array}$$

The medium ensures the client’s domain remains fixed through the entire interaction, regardless of whether the middleware chooses to interact with the server. This showcases how our medium transparently manages domain migration of participants.

Characterization Results We state results that offer a sound and complete account of the relationship between: (i) a global type $G$ (and its local types), (ii) its medium process $\text{Med}[G](\bar{c})$, and (iii) process implementations for the participants $\{p_1, \ldots, p_n\}$ of $G$. In a nutshell, these results say that the typeful composition of $\text{Med}[G](\bar{c})$ with processes for each
\( p_1, \ldots, p_n \) (well-typed in the system of §3) performs the intended global type. Crucially, these processes reside in distinct domains and can be independently developed, guided by their local type—they need not know about the medium’s existence or structure. The results generalize those in [7] to the domain-aware setting. Given a global type \( G \) with \( \text{part}(G) = \{p_1, \ldots, p_n\} \), below we write \( \text{npart}(G) \) to denote the set of indexed names \( \{c_{p_1}, \ldots, c_{p_n}\} \). We define:

**Definition 4.9 (Compositional Typing).** We say \( \Omega; \Gamma; \Delta \vdash M^\omega[G](\tilde{c}) :: z : C \) is a compositional typing if: (i) it is a valid typing derivation; (ii) \( \text{npart}(G) \subseteq \text{dom}(\Delta) \); and (iii) \( C = 1 \).

A compositional typing says that \( M^\omega[G](\tilde{c}) \) depends on behaviors associated to each participant of \( G \); it also specifies that \( M^\omega[G](\tilde{c}) \) does not offer any behaviors of its own.

The following definition relates binary session types and local types: the main difference is that the former do not mention participants. Below, \( B \) ranges over base types (\( \text{bool}, \text{nat}, \ldots \)).

**Definition 4.10 (Local Types \( \rightarrow \) Binary Types).** Mapping \( \llbracket \cdot \rrbracket \) from local types \( T \) (Def. 4.1) into binary types \( A \) (Def. 3.1) is inductively defined as \( \llbracket \text{end} \rrbracket = \llbracket B \rrbracket = 1 \) and

\[
\begin{align*}
\llbracket \text{proj}(i, U_i, T_i)_{i \in I} \rrbracket & = \llbracket \forall i : \llbracket U_i \rrbracket \otimes \llbracket T_i \rrbracket \rrbracket_{i \in I} \\
\llbracket \text{proj}(i, U_i, T_i)_{i \in I} \rrbracket & = \llbracket \forall i : \llbracket U_i \rrbracket \otimes \llbracket T_i \rrbracket \rrbracket_{i \in I} \\
\llbracket \text{proj}(i, T) \rrbracket & = \llbracket \forall i : \llbracket T \rrbracket \rrbracket \llbracket \downarrow \alpha.T \rrbracket = \llbracket \downarrow \alpha.T \rrbracket
\end{align*}
\]

Our first characterization result ensures that well-formedness of a global type \( G \) guarantees the typability of its medium \( M^\omega[G](\tilde{c}) \) using binary session types. Hence, it ensures that multiparty protocols can be analyzed by composing the medium with independently obtained, well-typed implementations for each protocol participant. Crucially, the resulting well-typed process will inherit all correctness properties ensured by binary typability established in §3.

**Theorem 4.11 (Global Types \( \rightarrow \) Typed Mediums).** If \( G \) is WF with \( \text{part}(G) = \{p_1, \ldots, p_n\} \) then \( \Omega; \Gamma; c_{p_1}; \llbracket G[p_1] \rrbracket[\omega_1], \ldots, c_{p_n}; \llbracket G[p_n] \rrbracket[\omega_n] \vdash M^\omega[G](\tilde{c}) :: \llbracket 1[\omega_m] \rrbracket \) is a compositional typing, for some \( \Omega, \Gamma, \) with \( \tilde{\omega} = \omega_1, \ldots, \omega_n \). We assume that \( \omega_i \not< \omega_m \) for all \( i \in \{1, \ldots, n\} \) (the medium’s namespace is accessible by all), and that \( i \neq j \) implies \( \omega_i \neq \omega_j \).

The second characterization result, given next, is the converse of Theorem 4.11: binary typability precisely delineates the interactions that underlie well-formed multiparty protocols. We need an auxiliary relation on local types, written \( \preceq \), that relates types with branching and “here” type operators, which have silent process interpretations (cf. Figure 1 and [9]).

First, we have \( T_1 \preceq T_2 \) if there is a \( T' \) such that \( T_1 \sqcup T' = T_2 \) (cf. Def. 4.3). Second, we have \( T_1 \preceq T_2 \) if (i) \( T_1 = T' \) and \( T_2 = \downarrow \alpha.T' \) and \( \alpha \) does not occur in \( T' \); but also if (ii) \( T_1 = \downarrow \alpha.T' \) and \( T_2 = T' \llbracket \omega/\alpha \rrbracket \). (See [9] for a formal definition of \( \preceq \)).

**Theorem 4.12 (Well-Typed Mediums \( \rightarrow \) Global Types).** Let \( G \) be a global type (cf. Def. 4.1). If \( \Omega; \Gamma; c_{p_1}; A_1[\omega_1], \ldots, c_{p_n}; A_n[\omega_n] \vdash M^\omega[G](\tilde{c}) :: z : 1[\omega_m] \) is a compositional typing then \( \exists T_1, \ldots, T_n \) such that \( G[p_j] \preceq T_j \) and \( \llbracket T_j \rrbracket = A_j \), for all \( p_j \in \text{part}(G) \).

The above theorems offer a static guarantee that connects multiparty protocols and well-typed processes. They can be used to establish also dynamic guarantees relating the behavior of a global type \( G \) and that of its associated set of multiparty systems (i.e., the typeful composition of \( M^\omega[G](\tilde{c}) \) with processes for each of \( p_i \in \text{part}(G) \)). These dynamic guarantees can be easily obtained by combining Theorems 4.11 and 4.12 with the approach in [7].

5 Related Work

There is a rich history of works on the logical foundations of concurrency (see, e.g., [4, 27, 1, 3]), which has been extended to session-based concurrency by Wadler [47], Dal Lago and Di
Giamberardino [35], and others. Medium-based analyses of multiparty sessions were developed
in [7] and used in an account of multiparty sessions in an extended classical linear logic [14].

Two salient calculi with distributed features are the Ambient calculus [16], in which
processes move across ambient (abstractions of administrative domains), and the distributed
π-calculus (Dπ) [29], which extends the π-calculus with flat locations, local communication,
and process migration. While domains in our model may be read as locations, this is just one
specific interpretation; they admit various alternative readings (e.g., administrative domains,
security-related levels), leveraging the partial view of the domain hierarchy. Type systems
for Ambient calculi such as [15, 6] enforce security and communication-oriented properties in
terms of ambient movement but do not cover issues of structured interaction, central in our
work. Garralda et al. [25] integrate binary sessions in an Ambient calculus, ensuring that
session protocols are undisturbed by ambient mobility. In contrast, our type system ensures
that both migration and communication are safe and, for the first time in such a setting,
satisfy global progress (i.e., session protocols never jeopardize migration and vice-versa).

The multiparty sessions with nested protocols of Demangeon and Honda [19] include
a nesting construct that is similar to our new global type $p \, \text{moves} \, q \, \to \, w$ for $G_1 ; G_2$, which
also introduces nesting. The focus in [19] is on modularity in choreographic programming;
domains nor domain migration are not addressed. The nested protocols in [19] can have local
participants and may be parameterized on data from previous actions. We conjecture that
our approach can accommodate local participants in a similar way. Data parameterization
can be transposed to our logical setting via dependent session types [44, 46]. Asynchrony and
recursive behaviors can also be integrated by exploiting existing logical foundations [23, 45].

Balzer et al. [2] overlay a notion of world and accessibility on a system of shared session
types to ensure deadlock-freedom. Their work differs substantially from ours: they instantiate
accessibility as a partial-order, equip sessions with multiple worlds and are not conservative
wrt linear logic, being closer to partial-order-based typings for deadlock-freedom [34, 37].

6 Concluding Remarks

We developed a Curry-Howard interpretation of hybrid linear logic as domain-aware session
types. Present in processes and types, domain-awareness can account for scenarios where
domain information is only determined at runtime. The resulting type system features strong
correctness properties for well-typed processes (session fidelity, global progress, termination).
Moreover, by leveraging a parametric accessibility relation, it rules out processes that
communicate with inaccessible domains, thus going beyond the scope of previous works.

As an application of our framework, we presented the first systematic study of domain-
awareness in a multiparty setting, considering multiparty sessions with domain-aware migra-
tion and communication whose semantics is given by a typed (binary) medium process that
orchestrates the multiparty protocol. Embedded in a fully distributed domain structure, our
medium is shown to strongly encode domain-aware multiparty sessions; it naturally allows us
to transpose the correctness properties of our logical development to the multiparty setting.

Our work opens up interesting avenues for future work. Mediums can be seen as monitors
that enforce the specification of a domain-aware multiparty session. We plan to investigate
contract-enforcing mediums building upon works such as [28, 33, 20], which study runtime
monitoring in session-based systems. Our enforcement of communication across accessible
domains suggests high-level similarities with information flow analyses in multiparty sessions
(cf. [13, 12, 17]), but does not capture the directionality needed to model such analyses
outright. It would be insightful to establish the precise relationship with such prior works.
References


