Reticulate: A Lattice Verification Tool for Session Types with Parallel Composition

Alexandre Zua Caldeira
LASIGE, Faculdade de Ciências, Universidade de Lisboa
Lisbon, Portugal
azcaldeira@fc.ul.pt

Vasco T. Vasconcelos
LASIGE, Faculdade de Ciências, Universidade de Lisboa
Lisbon, Portugal
vmvasconcelos@fc.ul.pt

Abstract

Session types specify communication protocols as types, ensuring that interacting parties follow an agreed sequence of message exchanges. We present Reticulate, a Python tool that constructs the state space of session types extended with a *parallel composition* constructor $S_1 \parallel S_2$, checks whether the resulting reachability poset forms a *lattice*, and classifies structure-preserving maps between state spaces. The parallel constructor forces a product lattice structure on the state space, making lattice verification both necessary and tractable. Reticulate handles recursion (via SCC quotients and topological-order dynamic programming), checks well-formedness and termination at the AST level, and supports a hierarchy of morphisms — from isomorphism to Galois connection. We validate the tool on 15 benchmark protocols drawn from real-world systems (SMTP, OAuth 2.0, WebSocket, MCP, A2A) and classic session-type literature. All benchmarks produce lattices and pass self-isomorphism checks.

CCS Concepts: • **Software and its engineering → Formal software verification**; *Formal methods.*

Keywords: session types, lattice theory, parallel composition, protocol verification, state-space construction

1 Introduction

Session types [7] discipline communication by assigning types to interaction protocols: a session type prescribes the legal sequences of messages, branches, and selections that may occur on a communication channel or object interface. Since their introduction, session types have been applied to objects [5, 13], multiparty protocols [8], and real-world toolchains such as Mungo [9] and Scribble [14].

A fundamental limitation of existing session type systems for objects is the assumption of *linear ownership*: at any point in the protocol, exactly one client holds the object. This precludes modeling concurrent access patterns where multiple threads or agents interact with the same object simultaneously — a scenario that arises naturally in file I/O, database connections, AI agent protocols, and network servers.

We address this limitation with the *parallel composition* constructor $S_1 \parallel S_2$, which splits a protocol into two concurrent sub-protocols. Methods from S_1 and S_2 may interleave freely until both sub-protocols reach their terminal state, at which point execution resumes sequentially. This constructor was introduced in the formal specification for Bica Reborn [5], the successor to the original Bica tool for session types on Java objects.

The lattice insight. The state space of a session type — the set of reachable protocol states ordered by reachability — forms a partially ordered set. Without parallel composition, the state space is a tree or DAG and the lattice structure is degenerate. With $\parallel$, the state space becomes the *product* of the component state spaces, ordered componentwise:

$$L(S_1 \parallel S_2) \;=\; L(S_1) \times L(S_2)$$

This product inherently has meets and joins, corresponding to fork and synchronization points. Lattice structure thus becomes *necessary* — not merely a pleasant mathematical property, but a structural invariant forced by the constructor.

Contributions. We present Reticulate, a Python library and CLI tool that:

1. Parses session types with branch (&), selection ($\oplus$), parallel ($\parallel$), recursion (μ), and sequencing.
2. Constructs labeled transition systems (state spaces) from session type ASTs, including product construction for $\parallel$.
3. Checks whether the reachability poset forms a lattice, handling cycles from recursion via SCC quotients.
4. Verifies well-formedness and termination of parallel branches at the AST level.
5. Classifies morphisms between state spaces: isomorphism, embedding, projection, homomorphism, and Galois connections.
6. Generates Hasse diagrams via Graphviz.

The tool is validated on 15 benchmark protocols with 477 tests.

Outline. Section 2 defines the session type grammar. Section 3 describes state-space construction. Section 4 presents the lattice verification algorithm. Section 5 covers well-formedness and termination. Section 6 introduces the morphism hierarchy. Section 7 evaluates the tool on 15 benchmarks. Section 8 describes the CLI and visualization. Section 9 discusses related work. Section 10 concludes.

2 Session Type Syntax

2.1 Grammar

Reticulate accepts session types in the following grammar, derived from the annotation syntax of Bica Reborn:

S	$::=$	$\&\{m_1 : S_1, \ldots, m_n : S_n\}$	branch (external choice)
	$\mid$	$\oplus\{l_1 : S_1, \ldots, l_n : S_n\}$	selection (internal choice)
	$\mid$	$S_1 \parallel S_2$	parallel composition
	$\mid$	$\mu X. S$	recursion
	$\mid$	X	type variable
	$\mid$	**end**	terminated
	$\mid$	$S_1 . S_2$	sequencing (sugar)

Branch ($\&$) represents external choice: the client selects a method m_i and the protocol continues as S_i. **Selection** ($\oplus$) represents internal choice: the object selects a label l_i and the protocol continues as S_i. **Parallel** ($\parallel$) forks the protocol into two concurrent sub-protocols. **Recursion** ($\mu X. S$) binds a variable X in the body S. **Sequencing** $m . S$ is syntactic sugar for a single-method branch: $m . S \equiv \&\{m : S\}$.

2.2 AST Representation

The parser produces an AST consisting of seven frozen dataclasses: `End`, `Var`, `Branch`, `Select`, `Parallel`, `Rec`, and `Sequence`. All nodes are immutable and hashable, enabling their use as dictionary keys and set elements.

The concrete syntax supports both ASCII (`||`, `+`) and Unicode (‖, ⊕, μ) variants. Two notations for parallel are accepted: `(S1 || S2)` and `||{S1, S2}`.

2.3 Running Example: SMTP

We use a simplified SMTP session as a running example throughout the paper:

```
connect . ehlo . rec X . &{
  mail: rcpt . data . ⊕{OK: X, ERR: X},
  quit: end
}
```

Listing 1. SMTP session type

This protocol has 7 states and 8 transitions: the client connects, sends EHLO, then loops sending mail (with RCPT/-DATA and OK or ERR responses) until quitting.

3 State-Space Construction

The state space of a session type S is a labeled transition system (LTS) $L(S) = (Q, \Sigma, \delta, q_\top, q_\bot)$ where Q is a finite set of states, Σ is an alphabet of method/label names, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_\top$ is the initial (top) state, and $q_\bot$ is the terminal (bottom) state.

3.1 Construction Rules

The state-space builder traverses the AST and generates states and transitions according to each constructor:

- **End**: returns the shared bottom state $q_\bot$.
- **Var(X)**: returns the state bound to X in the environment.
- **Branch** $\&\{m_i : S_i\}$: creates a fresh state with a transition labeled m_i to $L(S_i)$'s entry for each i.
- **Select** $\oplus\{l_i : S_i\}$: same structure as branch (transitions represent the object's choices).
- **Rec** $\mu X. S$: allocates a placeholder state for X, builds the body with X mapped to the placeholder, then merges the placeholder into the body's entry state — creating a back-edge (cycle).
- **Sequence** $S_1 . S_2$: builds S_2 first, then builds S_1 with the terminal state redirected to S_2's entry.
- **Parallel** $S_1 \parallel S_2$: delegates to the product construction (Section 3.2).

The builder maintains a monotonically increasing state counter and an environment mapping recursion variables to state IDs. After construction, only states reachable from $q_\top$ are retained.

3.2 Product Construction for Parallel

Given $S_1 \parallel S_2$, the tool builds each branch as an independent state space and computes their product:

Definition 3.1 (Product state space)**.** Let $L(S_1) = (Q_1, \Sigma_1, \delta_1, \top_1, \bot_1)$ and $L(S_2) = (Q_2, \Sigma_2, \delta_2, \top_2, \bot_2)$. The product state space is:

$$L(S_1 \parallel S_2) = Q_1 \times Q_2$$

with transitions:

$$(s_1, s_2) \xrightarrow{a} (s_1', s_2) \qquad \text{if } s_1 \xrightarrow{a} s_1' \text{ in } L(S_1)$$

$$(s_1, s_2) \xrightarrow{a} (s_1, s_2') \qquad \text{if } s_2 \xrightarrow{a} s_2' \text{ in } L(S_2)$$

Top is $(\top_1, \top_2)$; bottom is $(\bot_1, \bot_2)$.

Each component advances independently: a transition in S_1 changes only the first component, and vice versa. This models *unrestricted interleaving* of the two sub-protocols.

Figure 1 shows the product lattice for two three-state chains, producing the canonical 3×3 example from the formal specification.

4 Lattice Verification

The reachability relation on a state space defines a partial order: $s_1 \geq s_2$ iff there is a directed path from s_1 to s_2. The top element is the initial state (it reaches everything) and the bottom element is the terminal state (everything reaches it). We check whether this poset is a lattice.

4.1 Handling Cycles: SCC Quotients

Recursive session types introduce cycles (via the placeholder-merge construction for μ). Cycles violate antisymmetry: if s_1 reaches s_2 and s_2 reaches s_1, then $s_1 \geq s_2$ and $s_2 \geq s_1$ but $s_1 \neq s_2$. We resolve this by *quotienting*: states in the same strongly connected component (SCC) are identified.

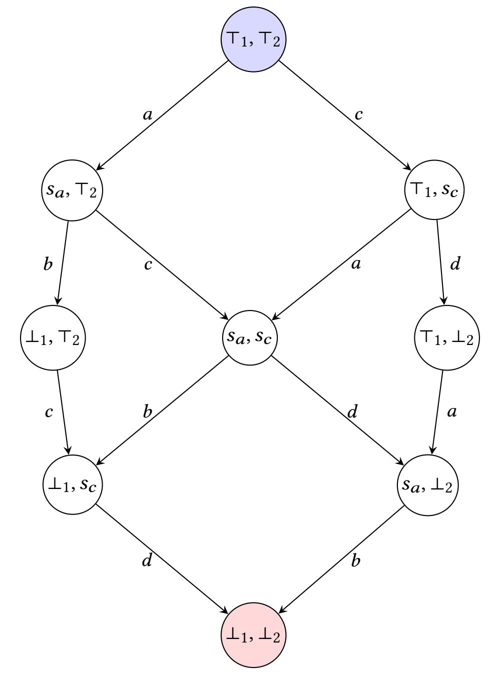

Figure 1. Product lattice for $a\,.\,b\,.\,\textbf{end} \parallel c\,.\,d\,.\,\textbf{end}$: a $3 \times 3 = 9$ state lattice. The top state $(\top_1, \top_2)$ is the fork point; the bottom state $(\bot_1, \bot_2)$ is the join point.

1. **Compute SCCs** using an iterative implementation of Tarjan's algorithm [11], producing SCCs in reverse topological order.
2. **Build quotient DAG**: collapse each SCC to a single node; edges between SCCs become edges in the DAG.
3. **Compute reachability** via topological-order dynamic programming: process nodes from sinks to sources for forward reachability, sources to sinks for reverse reachability.

4.2 Lattice Check

On the quotient DAG, we verify four conditions:

1. **Top**: the SCC containing $q_\top$ reaches all other SCCs.
2. **Bottom**: all SCCs reach the SCC containing $q_\bot$.
3. **All meets exist**: for every pair of quotient nodes a, b, the set of common lower bounds $\{c \mid a \geq c \text{ and } b \geq c\}$ has a greatest element (their meet $a \wedge b$).
4. **All joins exist**: for every pair a, b, the set of common upper bounds has a least element (their join $a \vee b$).

The meet computation finds lower bounds as the intersection of forward reachability sets, then selects the unique greatest element (one that reaches all other lower bounds). Joins are computed dually.

Theorem 4.1 (Product of lattices is a lattice, cf. [3])**.** *If $L(S_1)$ and $L(S_2)$ are lattices (after SCC quotient), then $L(S_1 \parallel S_2) = L(S_1) \times L(S_2)$ is a lattice, with componentwise meets and joins:*

$$(s_1, s_2) \wedge (s_1', s_2') = (s_1 \wedge s_1',\ s_2 \wedge s_2') \qquad (s_1, s_2) \vee (s_1', s_2') = (s_1 \vee s_1',\ s_2 \vee s_2')$$

The tool verifies this theorem empirically for each benchmark, providing confidence in the construction even when recursive branches produce complex quotient structures.

4.3 Result Reporting

The lattice check returns a `LatticeResult` frozen dataclass containing: `is_lattice` (boolean verdict), `has_top`, `has_bottom`, `all_meets_exist`, `all_joins_exist`, `num_scc` (number of SCCs in the quotient), and `counterexample` (a pair of states and the failing property, if any). This enables precise diagnostic messages when a state space fails lattice verification.

5 Well-Formedness and Termination

Well-formedness checking operates on the AST, before state-space construction. It ensures that parallel branches satisfy structural invariants required for the product construction to be sound.

5.1 Termination

A recursive session type $\mu X.\,S$ is *terminating* if the body S has at least one syntactic path to a leaf that is not X. This ensures that the recursion can eventually exit, reaching **end**.

Definition 5.1 (Exit path)**.** A node S has an *exit path* avoiding variable X if:

- $S = \textbf{end}$: trivially has an exit.
- $S = Y$ where $Y \neq X$: exits to an enclosing recursion.
- $S = \&\{m_i : S_i\}$ or $S = \oplus\{l_i : S_i\}$: some S_i has an exit path avoiding X.
- $S = S_1\,.\,S_2$ or $S = S_1 \parallel S_2$: both S_1 and S_2 have exit paths avoiding X.
- $S = \mu Y.\,S'$: the inner body S' has an exit path avoiding X (the inner binding Y is a different variable).

Node $S = X$ does *not* have an exit path (it loops back).

This check is decidable: it performs a single pass over the AST for each μ-binder encountered.

5.2 WF-Par: Well-Formedness of Parallel Composition

Each occurrence of $S_1 \parallel S_2$ must satisfy three conditions:

1. **Termination**: both S_1 and S_2 are terminating. This ensures the join point $(\textbf{end}, \textbf{end})$ is reachable.
2. **No cross-branch variables**: no recursion variable bound in S_1 occurs free in S_2, and vice versa. This ensures the branches can be built as independent state spaces.
3. **No nested parallel**: neither S_1 nor S_2 contains a $\parallel$ constructor. Nested parallelism is deferred to a future version.

The tool computes free and bound variable sets and checks for intersection. It reports structured error messages identifying which condition failed and which variables or subterms are involved.

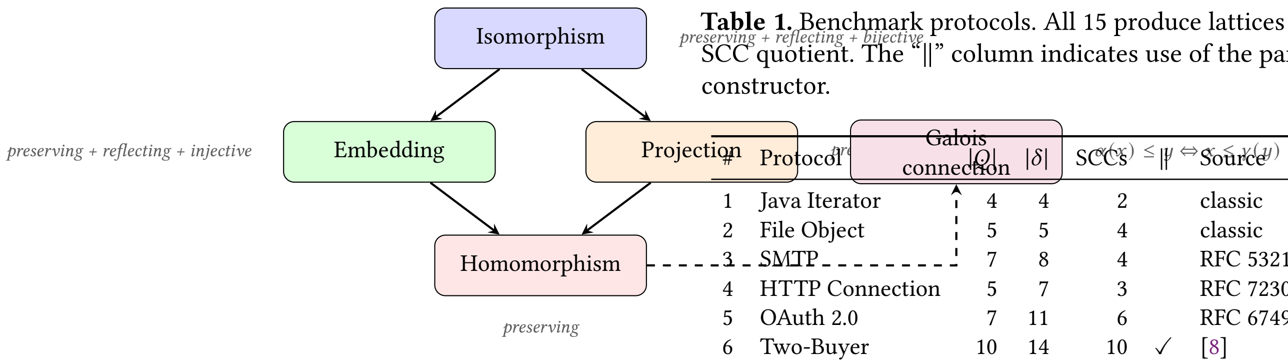
Isomorphism
preserving + reflecting + bijective
Embedding
preserving + reflecting + injective
Projection
Galois connection
$\alpha(x) \le y \Leftrightarrow x \le \gamma(y)$
Homomorphism
preserving

Figure 2. Morphism hierarchy. Arrows indicate subset relationships. Galois connections are related to homomorphisms but form a separate concept involving paired maps.

Table 1. Benchmark protocols. All 15 produce lattices after SCC quotient. The "∥" column indicates use of the parallel constructor.

#	Protocol	$\lvert Q\rvert$	$\lvert\delta\rvert$	SCCs	∥	Source
1	Java Iterator	4	4	2		classic
2	File Object	5	5	4		classic
3	SMTP	7	8	4		RFC 5321
4	HTTP Connection	5	7	3		RFC 7230
5	OAuth 2.0	7	11	6		RFC 6749
6	Two-Buyer	10	14	10	✓	[8]
7	MCP	7	17	5	✓	[1]
8	A2A	5	7	4		[6]
9	File Channel	7	12	7	✓	–
10	ATM	7	11	6		classic
11	Reentrant Lock	6	7	2		`j.u.c.locks`
12	WebSocket	6	15	6	✓	RFC 6455
13	DB Transaction	5	9	3		JDBC
14	Pub/Sub	4	7	3		classic
15	DNS Resolver	4	8	2		RFC 1035

6 Morphism Hierarchy

Reticulate classifies structure-preserving maps between state spaces, enabling comparison of protocols at the algebraic level.

6.1 Order Properties

Given state spaces $L(S)$ and $L(T)$ and a map $f : Q_S \to Q_T$:

- f is *order-preserving* if $s_1 \ge s_2$ implies $f(s_1) \ge f(s_2)$.
- f is *order-reflecting* if $f(s_1) \ge f(s_2)$ implies $s_1 \ge s_2$.

6.2 Classification

We classify morphisms by their structural properties:

Kind	Preserving	Reflecting	Injective	Surjective
Homomorphism	✓			
Projection	✓			✓
Embedding	✓	✓	✓	
Isomorphism	✓	✓	✓	✓

The hierarchy is ordered by strength: every isomorphism is an embedding, every embedding is a homomorphism, and every projection is a homomorphism.

6.3 Search Algorithms

Finding an isomorphism or embedding between state spaces is a constraint satisfaction problem. Reticulate uses backtracking search with pruning:

1. **Quick rejection**: compare state counts, transition counts, and degree-signature multisets.
2. **Anchor fixed points**: map $\top \mapsto \top$ and $\bot \mapsto \bot$.
3. **Signature pruning**: each state has a signature (out-degree, in-degree, reachability-set size); candidates must match.
4. **Compatibility check**: for each candidate assignment $s_1 \mapsto s_2$, verify order-preserving and order-reflecting constraints against the partial mapping.
5. **Full verification**: once all states are assigned, check the complete mapping.

6.4 Galois Connections

A pair of maps (α, γ) between state spaces S (concrete) and T (abstract) forms a *Galois connection* if:

$$\alpha(x) \le y \iff x \le \gamma(y) \qquad \forall\, x \in Q_S,\ y \in Q_T$$

This generalizes the abstraction–concretization framework of Cousot and Cousot [2] to session type state spaces. The tool checks the adjunction condition for all pairs.

6.5 Application: Product Commutativity

A direct application of isomorphism search is verifying product commutativity: $L(S_1 \parallel S_2) \cong L(S_2 \parallel S_1)$. The benchmarks (Section 7) confirm this for all protocols using ∥.

7 Benchmarks

We validate Reticulate on 15 benchmark protocols drawn from real-world systems and classic session-type literature. Table 1 summarizes the results.

7.1 Protocol Selection

The benchmarks span several categories:

- **Network protocols**: SMTP, HTTP, WebSocket, DNS — modeled from their respective RFCs.
- **Authentication**: OAuth 2.0 with its token lifecycle (use/refresh/revoke).
- **AI agent protocols**: MCP (Model Context Protocol) and A2A (Agent-to-Agent), representing emerging protocols for AI systems with concurrent tool execution and notification streams.

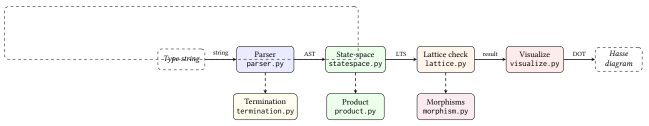

Figure 3. Reticulate pipeline. The CLI orchestrates the flow from type string through parsing, state-space construction, lattice verification, and visualization. Supporting modules (dashed) provide termination checking, product construction, and morphism analysis.

- **Classic session types**: Java Iterator, File Object, Two-Buyer, ATM — standard examples from the session-type literature.
- **Concurrency patterns**: File Channel, Reentrant Lock — modeling Java's `java.nio` and `java.util.concurrent.locks` APIs.
- **Data systems**: DB Transaction (JDBC-style), Pub/Sub (message broker client).

7.2 Results

All 15 benchmarks produce lattices after SCC quotient. Four benchmarks use the parallel constructor: Two-Buyer, MCP, File Channel, and WebSocket. These produce product lattices with 10, 7, 7, and 6 states respectively.

State-space sizes range from 4 states (Java Iterator, Pub/-Sub, DNS) to 10 (Two-Buyer). SCC counts range from 2 to 10; higher SCC counts occur in protocols with parallel composition, where the product construction does not introduce cycles.

Self-isomorphism ($L(S) \cong L(S)$) succeeds for all 15 benchmarks, validating the isomorphism search algorithm.

7.3 Validation Methodology

Each benchmark is defined as a `BenchmarkProtocol` dataclass specifying the session type string, expected state count, expected transition count, expected SCC count, and whether the protocol uses $\parallel$. The test suite verifies that Reticulate produces exactly these metrics, providing regression protection against changes to the construction algorithms.

8 Visualization and CLI

Reticulate provides a command-line interface and visualization pipeline.

8.1 CLI Modes

The tool is invoked as `python -m reticulate <type-string>` and supports three output modes:

- **Default**: prints a text summary — state count, transition count, SCC count, and lattice verdict.
- **`--dot`**: emits DOT source to standard output, suitable for piping to Graphviz.
- **`--hasse`**: renders a Hasse diagram to a file (PNG, SVG, or PDF via `--fmt`).

Additional flags control label display (`--no-labels`, `--no-edge-labels`), diagram title (`--title`), and output format.

8.2 Hasse Diagrams

The visualization module generates Hasse diagrams as DOT graphs, with:

- Top and bottom states highlighted in color.
- SCC collapsing: cyclic states from recursion are merged into single nodes.
- Counterexample highlighting: if the lattice check fails, the offending pair is marked.
- Edge labels showing method/selection names.

DOT generation uses only the standard library; rendering to image formats requires the optional `graphviz` Python package.

9 Related Work

Session types for objects. Bica [5] introduced session types for Java objects with branch and selection constructors. Mungo [9] and StMungo [10] extended this line with typestate-based checking and integration with the Scribble protocol description language [14]. None of these tools support a parallel composition constructor for concurrent access to objects.

Session type theory. Honda et al. [7] introduced binary session types. Vasconcelos [12] provides a comprehensive treatment of session type fundamentals. Honda, Yoshida, and Carbone [8] developed multiparty session types. Gay and Hole [4] established subtyping for session types. Our work differs in focusing on the *lattice structure* of session type state spaces, rather than on type soundness or process-calculus semantics.

Lattices in program analysis. Cousot and Cousot [2] established the connection between lattice theory and static analysis via abstract interpretation. Our Galois connection checking (Section 6) directly connects to this framework: a Galois connection between two session type state spaces is an abstraction–concretization pair in the abstract interpretation sense.

Lattice theory. Davey and Priestley [3] provide the standard reference for lattice theory. Our Theorem 4.1 (product of lattices is a lattice) is a classical result; the contribution here is the *application* to session types and the tool support for verification.

10 Conclusion

We presented Reticulate, a tool for constructing and verifying the lattice structure of session type state spaces extended with parallel composition. The parallel constructor $S_1 \parallel S_2$

forces a product lattice structure on the state space, connecting session type theory to lattice theory in a concrete, tool-supported way.

Reticulate is implemented in Python 3.12 with 8 modules, 477 tests, and 15 benchmark protocols. It provides a complete pipeline from parsing through state-space construction, lattice verification, morphism classification, and visualization.

Future work. Several directions remain:

- **Thread safety checking**: the formal specification defines concurrency levels and compatibility checks for methods in parallel branches; implementing this requires Java bytecode analysis (planned for Bica Reborn).
- **Subtyping**: extending the lattice framework to support session type subtyping with parallel composition, including the rules Sub-Par and Sub-Par-Safety from the specification.
- **Bisimulation**: connecting the morphism hierarchy to behavioral equivalences (bisimulation, trace equivalence) on session types.
- **Bica Reborn**: a Java annotation-based session type checker for objects, using Reticulate as the lattice verification back-end.
- **Nested parallelism**: extending the product construction and well-formedness checks to support $\|$ inside $\|$ branches.

References

[1] Anthropic. 2024. *Model Context Protocol Specification.* Technical Report. Anthropic. https://modelcontextprotocol.io/specification.

[2] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1977), 238–252. https://doi.org/10.1145/512950.512973

[3] B. A. Davey and H. A. Priestley. 2002. *Introduction to Lattices and Order* (2nd ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511809088

[4] Simon J. Gay and Malcolm Hole. 2005. Subtyping for Session Types in the Pi Calculus. *Acta Informatica* 42, 2–3 (2005), 191–225. https://doi.org/10.1007/s00236-005-0177-z

[5] Simon J. Gay, Vasco T. Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Zua Caldeira. 2010. Modular Session Types for Distributed Object-Oriented Programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 299–312. https://doi.org/10.1145/1706299.1706335

[6] Google. 2024. *Agent-to-Agent (A2A) Protocol.* Technical Report. Google. https://google.github.io/A2A/.

[7] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. *Lecture Notes in Computer Science* 1381 (1998), 122–138. https://doi.org/10.1007/BFb0053567

[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* ACM, 273–284. https://doi.org/10.1145/1328438.1328472

[9] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2016. Typechecking Protocols with Mungo and StMungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP).* ACM, 146–159. https://doi.org/10.1145/2967973.2968595

[10] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2017. Typechecking Protocols with Mungo and StMungo: A Session Type Toolchain for Java. In *Behavioural Types: from Theory to Tools.* River Publishers, 371–395.

[11] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. In *SIAM Journal on Computing*, Vol. 1. 146–160. https://doi.org/10.1137/0201010

[12] Vasco T. Vasconcelos. 2012. Fundamentals of Session Types. *Information and Computation* 217 (2012), 52–70. https://doi.org/10.1016/j.ic.2012.05.002

[13] Vasco T. Vasconcelos, Simon J. Gay, and António Ravara. 2006. Type Checking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science* 368, 1–2 (2006), 64–87. https://doi.org/10.1016/j.tcs.2006.06.028

[14] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *Trustworthy Global Computing (TGC).* Springer, 22–41. https://doi.org/10.1007/978-3-319-05119-2_3