Buffered Communication Analysis in Distributed Multiparty Sessions[⋆]

Pierre-Malo Deniélou and Nobuko Yoshida

Department of Computing, Imperial College London

Abstract. Many communication-centred systems today rely on asynchronous messaging among distributed peers to make efficient use of parallel execution and resource access. With such asynchrony, the communication buffers can happen to grow inconsiderately over time. This paper proposes a static verification methodology based on multiparty session types which can efficiently compute the upper bounds on buffer sizes. Our analysis relies on a uniform causality audit of the entire collaboration pattern — an examination that is not always possible from each end-point type. We extend this method to design algorithms that allocate communication channels in order to optimise the memory requirements of session executions. From these analyses, we propose two refinements methods which respect buffer bounds: a *global protocol refinement* that automatically inserts confirmation messages to guarantee stipulated buffer sizes and a *local protocol refinement* to optimise asynchronous messaging without buffer overflow. Finally our work is applied to overcome a buffer overflow problem of the multi-buffering algorithm.

1 Introduction

Session types for buffer bound analysis. The expensive cost of synchronous communications has led programmers to rely on asynchronous messaging for efficient network interactions. The downside is that non-blocking IO requires buffers that can grow inconsiderately over time, bringing systems to stop. The analysis and debugging of this phenomenon is mainly done by a tedious monitoring of the communicated messages of the whole distributed system. This paper shows that, when a global interaction pattern is explicitly specified as a *multiparty session* [1, 11, 15, 22], types can provide an effective way to statically verify buffer usage and communication optimisations, automatically guaranteeing safe and deadlock-free runs.

Session types, first introduced in [10, 20], can specify communication protocols by describing the sequences and types of read, write and choices on a given channel. For example, type $T_0 = !\langle \mathsf{nat} \rangle; !\langle \mathsf{string} \rangle; ?\langle \mathsf{real} \rangle; \mathsf{end}$, in the original binary session type syntax, expresses that a nat-value and string-value will be sent in that order, then that a real-value is expected as an input, and finally that the protocol ends.

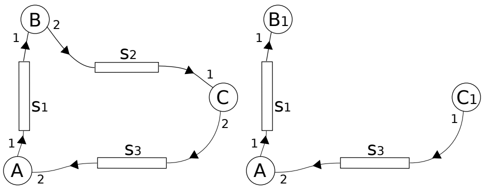
We can use session types to calculate the upper bounds of the buffer sizes of asynchronous channels (message passing is non-blocking and order-preserving using FIFO buffers). For example, from type T_0, we can compute that the maximum number of messages that might be stored in a communication buffer is two, while a different type

[⋆] The work is partially supported by EPSRC EP/F003757/01 and EPSRC G015635/01.

$T_1 = !\langle \mathsf{nat}\rangle; ?\langle \mathsf{real}\rangle; !\langle \mathsf{string}\rangle; \mathsf{end}$ guarantees a maximum size of one, since the dual process engaged with T_1 is forced to consume a nat-value before sending the next real-message. This use of session types is informally observed in [7] and formally studied in [8] for binary session types. However, the binary case does not yield a direct extension to multiparty interactions as explained below.

Buffer bounds analysis in multiparty sessions. We start by illustrating the difficulties of such an analysis on a simple three party interaction (Example (a) below), where $s!\langle V\rangle$ is an output of V to s, $s?(x);P$ an input at s, and $\mu X.P$ a recursive agent:

Example (a)
(A) Alice=$\mu X.s_1!\langle 1\rangle;s_3?(x);X$
(B) Bob =$\mu X.s_1?(x);s_2!\langle \mathit{Orange}\rangle;X$
(C) Carol=$\mu X.s_2?(x);s_3!\langle 2.4\rangle;X$
Example (b)
(B$_1$) Bob$_1$ =$\mu X.s_1?(x);X$
(C$_1$) Carol$_1$=$\mu X.s_3!\langle 2.4\rangle;X$



Example (a) **Example (b)**

We assume the three buffers of s_1, s_2 and s_3 are initially empty and that values are pushed and read one by one. Assuming session types ensure that accesses to buffers do not create any race condition at any moment of the infinite protocol execution, none of the channels s_1, s_2, s_3 need to buffer more that one value at any given time.

However, if we change Bob and Carol to Bob$_1$ and Carol$_1$ as Example (b) above, while they still interact correctly, the buffers of s_1 and s_3 need an unbounded size because of the *lack of synchronisation* between Bob$_1$ and Carol$_1$.

The main difficulty of the communication buffer analysis is that, unlike in binary session types, each end-point type itself does not provide enough information: for example, Alice's local type $T_a = \mu \mathbf{x}.s_1!\langle \mathsf{nat}\rangle; s_3?\langle \mathsf{real}\rangle; \mathbf{x}$ (repeatedly sends a nat-value to s_1 and receives a real-value from s_3) is *the same* in both Examples (a) and (b), while the needed buffer size for s_1 and s_3 are different (1 in (a) and ∞ in (b)) due to the change in the other parties' behaviours. Our first question is: *can we statically and efficiently determine the upper size of buffers in multiparty interactions?* In our case, we take advantage of the existence of a global session type [1, 11, 15, 22] for the analysis:

$$G \;=\; \mu \mathbf{x}.\mathrm{Alice} \to \mathrm{Bob}\colon\, s_1\,\langle \mathsf{nat}\rangle; \mathrm{Bob} \to \mathrm{Carol}\colon\, s_2\,\langle \mathsf{string}\rangle; \mathrm{Carol} \to \mathrm{Alice}\colon\, s_3\,\langle \mathsf{real}\rangle; \mathbf{x}$$

The above type represents the global interaction between Alice-Bob-Carol in (a) where Alice → Bob: $s_1\,\langle \mathsf{nat}\rangle$; means that Alice sends a nat-value to Bob through buffer s_1. To analyse buffer usage, we consider sessions as graphs and track *causal chains* for each channel: alternated message production and consumption mark the execution points at which buffers are emptied. This can be observed in Example (a). On the other hand, the global type of Alice-Bob$_1$-Carol$_1$ in (b) lacks the second Bob → Carol: no message forces Carol to wait for Bob's reception before sending the next message. In that case, each buffer may accumulate an unbounded number of messages.

Channel allocation. Our next problem is about resource allocation. *Given a global scenario, can we assign the minimum number of resources (channels) without conflict* so that, for instance, we can efficiently open a minimal number of sockets for a given network interaction? Assume Alice and Carol in (a) wish to communicate one more message after completing three communications, where the new communication happens on a fresh channel s_4 (Example (c) below). Can we reuse either s_1, s_2 or s_3 for this new communication? Reusing s_1 creates a writing conflict (the order between Alice's first and third messages would be unspecified) and reusing s_3 would create a reading conflict (Carol could read her own message).

Example (c)

(A_2) $\text{Alice}_2 = \mu X.s_1!\langle 1\rangle;s_3?(x);s_4!\langle x+1\rangle;X$

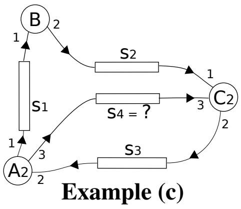
(B) $\text{Bob} = \mu X.s_1?(x);s_2!\langle \mathit{Orange}\rangle;X$

(C_2) $\text{Carol}_2 = \mu X.s_2?(x);s_3!\langle 2.4\rangle;s_4?(y);X$

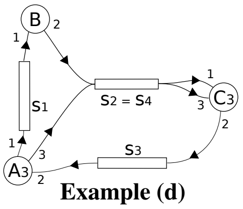
Example (d)

(A_3) $\text{Alice}_3 = \mu X.s_1!\langle 1\rangle;s_3?(x);s_2!\langle x+1\rangle;X$

(C_3) $\text{Carol}_3 = \mu X.s_2?(x);s_3!\langle 2.4\rangle;s_2?(y);X$



Example (c)



Example (d)

The only safe way to reuse a channel in Example (c) is to merge s_2 and s_4 as in Example (d), in which case communications on other channels prevent any conflict.

Global refinement for multiparty sessions. The third issue is how to fix a buffer overflow problem by "global refinement", i.e. alteration of the original global protocol to satisfy given buffer sizes. Here, our simple approach is the insertion of a minimal number of *confirmation messages* to enforce synchronisation. In network or business protocols, they can be implemented as a system level signal. Consider the interaction (b) among Alice-Bob$_1$-Carol$_1$ where each buffer requires an unbounded size. If we wish to enforce a buffer size of at most 2, we can build a new global type where one confirmation message from Bob to Carol is inserted in any second iteration as:

$$\begin{aligned} G' \ = \ \mu\mathbf{x}.\ & \text{Alice} \to \text{Bob}\colon s_1\,\langle\mathsf{nat}\rangle;\text{Carol} \to \text{Alice}\colon s_3\,\langle\mathsf{real}\rangle; \\ & \text{Alice} \to \text{Bob}\colon s_1\,\langle\mathsf{nat}\rangle;\text{Bob} \to \text{Carol}\colon s_2\,\langle\mathsf{string}\rangle;\text{Carol} \to \text{Alice}\colon s_3\,\langle\mathsf{real}\rangle;\mathbf{x} \end{aligned}$$

The revised processes following G' are given as:

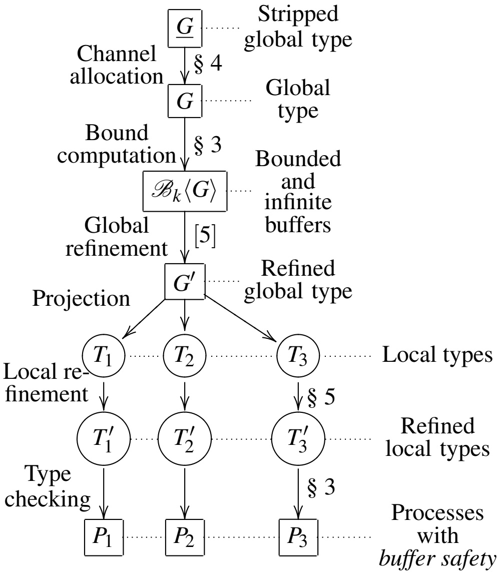
$$\text{Bob}_4 = \mu X.s_1?(x);s_1?(x);s_2!\langle \mathit{Signal}\rangle;X \qquad \text{Carol}_4 = \mu X.s_3!\langle 2.4\rangle;s_2?(x).s_3!\langle 2.4\rangle;X$$

Local refinement for multiparty messaging optimisations. The last issue is about flexible local refinement (optimisations) based on [14, 15]. Assume that, in Example (a), Bob wishes to always start the asynchronous transmission of the string *Orange* to the buffer s_3 without waiting for the delivery of the first nat-value from Alice on s_1.

$$\text{Bob}_5 = \mu X.s_2!\langle \mathit{Orange}\rangle;s_1?(x);X \tag{1.1}$$

Due to Bob's unilateral implementation change, all three minimal buffer sizes go up from 1 to 2. Moreover, suppose Bob repeatedly applies the same optimisation on his next n messages, as in $s_2!\langle \mathit{Orange}\rangle;s_2!\langle \mathit{Orange}\rangle;..;s_2!\langle \mathit{Orange}\rangle;\text{Bob}$. While the result is communication-safe (no mismatch of the communication with Carol), all three minimal buffer sizes go up from 1 to n. *How can we perform local optimisation without altering buffer sizes in a multiparty session*?

Contributions are summarised in the figure below. To the best of our knowledge, our work is the first which guarantees safe buffered multiparty communications for the π-calculus with communication-safety, progress and flexible refinements. The key contribution is a general causal analysis over graphs constructed from multiparty session types (§ 3). Due to the space limitation, we omit the global refinement. A full version which includes the global refinement, omitted definitions, examples and proofs, and a prototype for computing the upper bound on the buffer size are available [5].



1. The overall analysis starts from global types that have no channel annotation. We attribute channels based on memory requirements (§ 4).
2. From global types, our bound analysis computes the buffer bounds of finite channels and finds the infinite ones (§ 3)
3. The global refinement method then introduces additional messages to prevent any unboundedness (long version [5]).
4. Once the global type has been projected to local types, local refinement can optimise the distributed execution of the participants' processes (§ 5).
5. The running optimised processes can then be typed and enjoy communication, buffer and type safety and progress properties (§ 3).
6. We apply our work to the multibuffering algorithm (§ 5.1).

2 Asynchronous Multiparty Sessions

Syntax. We start from the π-calculus for multiparty sessions from [11] with unbounded and bounded buffers. Base sets and the grammars are given below.

P	$::=$	$\overline{a}_{[2..n]}(\tilde{s}^{\tilde{m}}).P \mid a_{[p]}(\tilde{s}).P$	request, accept	$a,b,x,y,..$	shared names
	$\mid$	$s!\langle\tilde{e}\rangle;P \mid s?(\tilde{x});P$	send, receive	$s,t,..$	session channels
	$\mid$	$s!\langle\!\langle\tilde{s}\rangle\!\rangle;P \mid s?((\tilde{s}));P$	session send, receive	$l,l',..$	labels
	$\mid$	$s\lhd l;P \mid s\rhd\{l_i\colon P_i\}_{i\in I}$	selection, branch	$X,Y,..$	process variables
	$\mid$	$\text{if } e \text{ then } P \text{ else } Q$	conditional	$m,n,..$	buffer size (integers or ∞)
	$\mid$	$\mathbf{0} \mid (\nu a)P \mid (\nu\tilde{s})P$	inact, hiding	$e ::= v \mid e \text{ and } e' \cdots$	expressions
	$\mid$	$P \mid Q \mid \mu X.P \mid X$	par, recursion	$v ::= a \mid \text{true} \mid \text{false} \cdots$	values
	$\mid$	$s^n\!:\!\tilde{h}$	message buffer	$h ::= l \mid \tilde{v} \mid \tilde{t}$	message values

$\overline{a}_{[2..n]}(\tilde{s}^{\tilde{m}}).P$ initiates, through a shared name a, a new session s_i with buffer size m_i ($1 \leq n \leq \infty$) with other participants, each of the form $a_{[p]}(\tilde{s}).Q$ with $1 \leq p \leq n-1$. The s_i in vector $\tilde{s}$ is a session channel (bounded by buffer size m_i) used in the session. We call p, q,... (natural numbers) the *participants* of a session. Session communications (which take place inside an established session) are performed by the sending and receiving of a value; the session sending and reception (where the former delegates to the latter the capability to participate in a session by passing a channel associated with the session which is called *delegation*); and by selection and branching (the former chooses one of the branches offered by the latter). $s^n\!:\!\tilde{h}$ is a *message buffer of size n* representing ordered messages in transit $\tilde{h}$ with destination s. This may be considered as a network pipe in a TCP-like transport with fixed bandwidth. The rest of the syntax is standard from [11]. We often omit n from $s^n:\tilde{h}$, $\mathbf{0}$, and unimportant arguments e.g. $s!\langle\rangle$ and $s?();P$. An *initial* process does not contain any runtime syntax (buffers and session hiding).

Reductions. A selection of reduction rules is given below.

$$\overline{a}[2..\mathtt{n}]\,(\tilde{s}^{\tilde{n}}).P_1 \mid a[2]\,(\tilde{s}).P_2 \mid \cdots \mid a[\mathtt{n}]\,(\tilde{s}).P_{\mathtt{n}} \;\rightarrow\; (\nu\,\tilde{s})(P_1 \mid P_2 \mid \ldots \mid P_{\mathtt{n}} \mid s_1^{n_1}:\emptyset \mid \ldots \mid s_m^{n_m}:\emptyset)$$

$$s!\,\langle\tilde{e}\rangle;P \mid s^n:\tilde{h} \;\rightarrow\; P \mid s^n:\tilde{h}\cdot\tilde{v}\;\;(n \gnsim |\tilde{h}|,\; e_i \downarrow v_i) \qquad s?(\tilde{x});P \mid s^n:\tilde{v}\cdot\tilde{h} \rightarrow P[\tilde{v}/\tilde{x}] \mid s^n:\tilde{h}$$

$$s!\langle\!\langle\tilde{t}\rangle\!\rangle;P \mid s^n:\tilde{h} \;\rightarrow\; P \mid s^n:\tilde{h}\cdot\tilde{t}\;\;(n \gnsim |\tilde{h}|) \qquad s?(\!(\tilde{t})\!);P \mid s^n:\tilde{t}\cdot\tilde{h} \rightarrow P \mid s^n:\tilde{h}$$

$$s \lhd l;P \mid s^n:\tilde{h} \;\rightarrow\; P \mid s^n:\tilde{h}\cdot l\;\;(n \gnsim |\tilde{h}|) \qquad s \rhd \{l_i\colon P_i\}_{i\in I} \mid s^n:l_j\cdot\tilde{h} \rightarrow P_j \mid s^n:\tilde{h} \qquad (j \in I)$$

The first rule describes the initiation of a new session among $\mathtt{n}$ participants that synchronise over the shared name a. After the initiation, they will share m fresh private session channels s_i and the associated m empty buffers of size n_m ($\emptyset$ denotes an empty queue). The output rules for values, sessions and selection respectively enqueue values, sessions and labels if the buffer is not full. $e_i \downarrow v_i$ denotes the evaluation of e_i to v_i. We define $|\emptyset| = 0$ and $|\tilde{h}\cdot h| = |\tilde{h}| + 1$. The size $n = \infty$ corresponds to the original asynchronous unbounded buffered semantics [11]. The input rules perform the complementary operations. Processes are considered modulo a structural equivalence $\equiv$, whose definition is standard (e.g. $\mu X.P \equiv P[\mu X.P/X]$) [11].

3 Bound Analysis in Multiparty Sessions

This section presents an analysis of causal chains and buffer sizes and introduces the typing system for the *buffer safety* property (Corollary 3.9).

3.1 Global Types and Dependencies

Global types. A *global type*, written by $G, G', ..,$ describes the whole conversation scenario of a multiparty session as a type signature. Our starting syntax is from [11].

$$\begin{array}{rcll}
G, G' & ::= & \mathtt{p} \rightarrow \mathtt{p}'\colon k\,\langle U\rangle; G' & \text{values} \\
& \mid & \mathtt{p} \rightarrow \mathtt{p}'\colon k\,\{l_j\colon G_j\}_{j\in J} & \text{branching} \\
& \mid & \mu\mathbf{x}.G \mid \mathbf{x} \mid \mathsf{end} & \text{recursion, end}
\end{array}
\qquad
\begin{array}{rcll}
U, U' & ::= & \tilde{S} \mid T@\mathtt{p} & \text{sorts, session} \\
S, S' & ::= & \mathsf{bool} \mid \mathsf{nat} \mid G & \text{base, shared}
\end{array}$$

Type $\mathtt{p} \rightarrow \mathtt{p}'\colon k\,\langle U\rangle; G'$ says that participant $\mathtt{p}$ sends a message of type U on the channel k (represented as a natural number) so that participant $\mathtt{p}'$ can receive it. The session continues with the interactions described in G'. The *value types* U, U' are either a vector of sorts or a *located type* $T@\mathtt{p}$, representing a local type T assigned to participant $\mathtt{p}$. Located types are used for delegation and defined in § 3.3. *Sorts* S, S' are either base types or global types for shared names. Type $\mathtt{p} \rightarrow \mathtt{p}'\colon k\,\{l_j\colon G_j\}_{j\in J}$ says that participant $\mathtt{p}$ can invoke one of the l_i labels on channel k (for participant $\mathtt{p}'$ to read) and that interactions described in G_j follow. We require $\mathtt{p} \neq \mathtt{p}'$ to prevent self-sent messages. Type $\mu\mathbf{x}.G$ is for recursive protocols, assuming the type variables $(\mathbf{x}, \mathbf{x}', \ldots)$ are guarded in the standard way, i.e. they only occur under values or branchings. We assume G in value types is closed, i.e. without free type variables. Type end represents session termination (often omitted). $k \in G$ means k appears in G. The functions $chans(G)$ and $prins(G)$ respectively give the number of channels and participants of G.

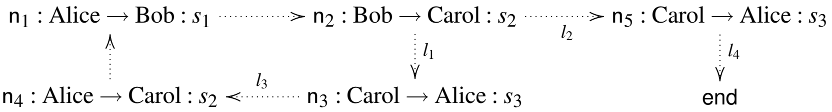
Sessions as graphs. Global types can be seen (isomorphically) as *session graphs*, that we define in the following way. First, we annotate in G each syntax occurrence of subterms of the form $\mathtt{p} \to \mathtt{p}'\colon k\,\langle U\rangle;G'$ or $\mathtt{p} \to \mathtt{p}'\colon k\,\{l_j\colon G_j\}_{j\in J}$ with a node name $(\mathsf{n}_1,\mathsf{n}_2,\ldots)$. Then, we inductively define a function $\mathtt{node}_G$ that gives a node n_k (or the special node end) for each of the syntactic subterm of G as follows:

$$\begin{array}{rlrl}
\mathtt{node}_G(\mathsf{end}) = \mathsf{end} & & \mathtt{node}_G(\mathsf{n}_i : \mathtt{p} \to \mathtt{p}'\colon k\,\langle U\rangle;G') & = \mathsf{n}_i \\
\mathtt{node}_G(\mu\mathbf{x}.G') = \mathtt{node}_G(G') & & \mathtt{node}_G(\mathsf{n}_j : \mathtt{p} \to \mathtt{p}'\colon k\,\{l_j\colon G_j\}_{j\in J}) & = \mathsf{n}_j \\
\mathtt{node}_G(\mathbf{x}) = \mathtt{node}_G(\mu\mathbf{x}.G') & \multicolumn{3}{l}{\textit{(if the binder of } \mathbf{x} \textit{ is } \mu\mathbf{x}.G' \in G\textit{)}}
\end{array}$$

We define G as a session graph in the following way: for each subterm of G of the form $\mathsf{n} : \mathtt{p} \to \mathtt{p}'\colon k\,\langle U\rangle;G'$, we have an edge from n to $\mathtt{node}_G(G')$, and for each subterm of G of the form $\mathsf{n}' : \mathtt{p} \to \mathtt{p}'\colon k\,\{l_j\colon G_j\}_{j\in J}$, we have edges from n' to each of the $\mathtt{node}_G(G_j)$ for $j \in J$. We also define the functions $\mathtt{pfx}(\mathsf{n}_i)$ and $\mathtt{ch}(\mathsf{n}_i)$ that respectively give the prefix ($\mathtt{p} \to \mathtt{p}' : k$) and channel ($k$) that correspond to n_i. For a global type G, $\mathtt{node}_G(G)$ distinguishes the *initial* node. $\mathit{size}(G)$ denotes the number of edges of G.

Example 3.1 (Session graph) Our running example extends Example (a) from § 1 with branching. Below, we give the global type followed by its graph representation, with the edges as the dotted arrows (labels are for information). n_1 is the initial node.

$$\mu\mathbf{x}.\text{Alice} \to \text{Bob}\colon s_1\,\langle \mathsf{nat}\rangle;\text{Bob} \to \text{Carol} : s_2 \left\{ \begin{array}{l} l_1 : \text{Carol} \to \text{Alice} : s_3\{l_3 : \text{Alice} \to \text{Carol}\colon s_2\,\langle \mathsf{string}\rangle;\mathbf{x}\} \\ l_2 : \text{Carol} \to \text{Alice} : s_3\{l_4 : \mathsf{end}\} \end{array} \right\}$$



The recursion call yields a cycle in the graph, while branching gives the edges l_1 and l_2.

The edges of a given session graph G define a successor relation between nodes, written $\mathsf{n} \prec \mathsf{n}'$ (omitting G). Paths in this session graph are referred to by the sequence of nodes they pass through: a path $\mathsf{n}_0 \prec \ldots \prec \mathsf{n}_n$ can be written more concisely $\mathsf{n}_0 \ldots \mathsf{n}_n$ or $\tilde{\mathsf{n}}$ when there is no ambiguity. We say that a path $\mathsf{n}_0 \ldots \mathsf{n}_n$ *has suffix* $\mathsf{n}_i \ldots \mathsf{n}_n$ for $0 < i < n$. The empty path is ε. The transitive closure of $\prec$ is $\prec\!\!\prec$.

IO-chains. We detect causality chains in a given G by the relation $\prec_{\mathtt{IO}}$, defined below:

$$\mathsf{n}_1 \prec_{\mathtt{IO}} \mathsf{n}_2 \;\text{ if } \mathsf{n}_1 \prec\!\!\prec \mathsf{n}_2 \text{ and } \mathtt{pfx}(\mathsf{n}_1) = \mathtt{p}_1 \to \mathtt{p} : k_1 \text{ and } \mathtt{pfx}(\mathsf{n}_2) = \mathtt{p} \to \mathtt{p}_2 : k_2 \text{ with } k_1 \neq k_2$$

The relation $\prec_{\mathtt{IO}}$ asserts the order between a reception by a principal and the next message it sends. An *input-output dependency (IO-dependency)* from n_1 to n_n is a chain $\mathsf{n}_1 \prec_{\mathtt{IO}} \cdots \prec_{\mathtt{IO}} \mathsf{n}_n$ ($n \geq 1$).

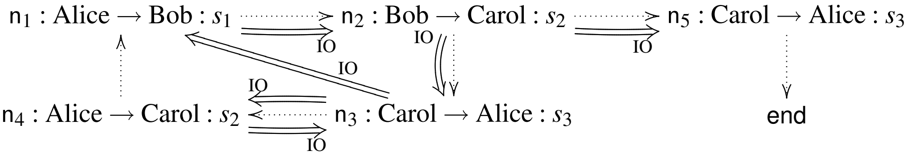
3.2 Algorithms for Buffer Size Analysis

Unbounded buffers. In some sessions, messages (sent asynchronously) can accumulate without moderation in a buffer. A simple test can predict which channels require an unbounded buffer. We use the fact that IO-dependencies characterise the necessity for a channel buffer to be emptied before proceeding. Infinite channels are the ones where such a dependency is missing.

Definition 3.2 (infinite and finite) A channel k is said to be *finite* if, for every node $\mathsf{n} \in G$ and for every cycle $\tilde{\mathsf{n}}$ for $\prec$ such that $\mathtt{ch}(\mathsf{n}) = k$ and $\mathsf{n} \in \tilde{\mathsf{n}}$, there exists a cycle for $\prec_{\mathtt{IO}}$ that starts from n and only involves nodes from $\tilde{\mathsf{n}}$. The other channels are *infinite*.

Correspondingly, buffers are said to be *bounded* or *unbounded*. Given G, checking for the infinity of k in G can be computed in a time bounded by $O(size(G)^3)$. The proof relies on the fact that establishing all IO-dependencies of a given session has $O(size(G)^3)$ time-complexity (assuming the number of participants as a constant).

Example 3.3 (Session graph and infinite channels) We illustrate on our running example the previous notions. We add to the picture the IO-dependencies (with $\Rightarrow$).



n_1 : Alice → Bob : s_1
n_2 : Bob → Carol : s_2
n_5 : Carol → Alice : s_3
n_4 : Alice → Carol : s_2
n_3 : Carol → Alice : s_3
end
IO
IO
IO
IO
IO
IO

Since each node of the main cycle $\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3\mathsf{n}_4$ is part of the IO-cycles $\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3$ or $\mathsf{n}_3\mathsf{n}_4$, there are no infinite channels in this session.

Counting finite buffer size. To compute the bounds on buffer sizes, we first need to define a property on paths that characterises when a buffer has to be emptied.

Definition 3.4 (reset) If $\tilde{\mathsf{n}} = \mathsf{n}_0 \ldots \mathsf{n}_n\mathsf{n}$ is a path in G, the property Reset$(\tilde{\mathsf{n}})$ holds if there exist $0 \leq i_0 < \ldots < i_j \leq n$ $(j \geq 1)$ such that $\mathsf{n}_{i_0} \prec_{\mathtt{IO}} \ldots \prec_{\mathtt{IO}} \mathsf{n}_{i_j} \prec_{\mathtt{IO}} \mathsf{n}$ and $\mathtt{ch}(\mathsf{n}_{i_0}) = \mathtt{ch}(\mathsf{n})$. One practical instance of the nodes $\{\mathsf{n}_{i_0}, \ldots, \mathsf{n}_{i_j}, \mathsf{n}\}$ is called the reset nodes of $\tilde{\mathsf{n}}$.

The paths that satisfy the reset property are the ones for which there exists a reception guard to the last node.

Now that we know which buffers are infinite and have characterised the resetting paths that control buffer growth, we can describe our algorithm to count the buffer size required by finite channels. For each channel k of a global session type G, we define a function $\mathscr{B}_k\langle G\rangle$ that will compute the bound on the buffer size of channel k. The key step is to reset the counter when we recognise the appropriate IO-dependencies.

Definition 3.5 (bound computation) Given a session graph G, for each channel k, we compute the bound as $\mathscr{B}_k\langle G\rangle = \mathscr{B}_k\langle 0, \emptyset, \varepsilon, \mathsf{n}_0\rangle$ for n_0 the initial node of G.

$$\mathscr{B}_k\langle m, \mathscr{P}, \tilde{\mathsf{n}}, \mathsf{n}\rangle = \begin{cases} 0 & \text{if } \mathsf{n} = \mathsf{end} \text{ or } \tilde{\mathsf{n}} \in \mathscr{P} \\ \max_{\mathsf{n} \prec \mathsf{n}'} \mathscr{B}_k\langle m, \{\tilde{\mathsf{n}}\} \cup \mathscr{P}, \tilde{\mathsf{n}}\mathsf{n}, \mathsf{n}'\rangle & \text{if } \mathtt{ch}(\mathsf{n}) = k', k \neq k' \\ \max_{\mathsf{n} \prec \mathsf{n}'} \mathscr{B}_k\langle 1, \{\tilde{\mathsf{n}}\} \cup \mathscr{P}, \mathsf{n}, \mathsf{n}'\rangle & \text{if } \mathtt{ch}(\mathsf{n}) = k, \text{Reset}(\tilde{\mathsf{n}}\mathsf{n}) \\ \max(m+1, \max_{\mathsf{n} \prec \mathsf{n}'} \mathscr{B}_k\langle m+1, \{\tilde{\mathsf{n}}\} \cup \mathscr{P}, \tilde{\mathsf{n}}\mathsf{n}, \mathsf{n}'\rangle) & \text{if } \mathtt{ch}(\mathsf{n}) = k, \neg\text{Reset}(\tilde{\mathsf{n}}\mathsf{n}) \end{cases}$$

The algorithm explores all the paths of the session graph until they grow to satisfy the reset property. Since we examine only finite channels, the length of such paths is limited and the algorithm terminates. The bound on the buffer size of a channel is the maximum buffer size required over these paths. For each path, the algorithm acts recursively on the edges and maintains a counter (m in $\mathscr{B}_k\langle m, \mathscr{P}, \tilde{\mathsf{n}}, \mathsf{n}\rangle$) that records the current size of the buffer. If the current prefix does not involve the channel k, the size of the buffer is unchanged and the computation continues to the next nodes. If the current prefix uses the channel k, there are two cases: (a) the reset property holds for the current path, in which case the buffer has been emptied prior to the current message; or (b) the reset property does not hold and the buffer needs to be able to keep one more value. When there are no further node, or when the path currently examined has already been considered (i.e. is in $\mathscr{P}$), the algorithm stops.

Given a global type G, the upper bound of channel k in G can be computed in polynomial time. Note that the computation can be done for all channels at once.

Example 3.6 (buffer bound analysis) We illustrate the algorithm on our running session example, where we compute the bound for channel s_2 (we omit $\mathscr{P}$ for readability):

$\mathscr{B}_{s_2}\langle 0,\varepsilon,\mathsf{n}_1\rangle$	max	explanation
$= \mathscr{B}_{s_2}\langle 0,\mathsf{n}_1,\mathsf{n}_2\rangle$	0	$s_1 \neq s_2$
$= \max(\mathscr{B}_{s_2}\langle 1,\mathsf{n}_1\mathsf{n}_2,\mathsf{n}_3\rangle,\mathscr{B}_{s_2}\langle 1,\mathsf{n}_1\mathsf{n}_2,\mathsf{n}_5\rangle)$	1	$\neg\mathrm{Reset}(\mathsf{n}_1\mathsf{n}_2),\neg\mathrm{Reset}(\mathsf{n}_1\mathsf{n}_3)$
$= \max(\mathscr{B}_{s_2}\langle 1,\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3,\mathsf{n}_4\rangle,\mathscr{B}_{s_2}\langle 1,\mathsf{n}_1\mathsf{n}_2\mathsf{n}_5,\mathsf{end}\rangle)$	1	$s_3 \neq s_2$
$= \max(\mathscr{B}_{s_2}\langle 1,\mathsf{n}_4,\mathsf{n}_1\rangle,0)$	1	$\mathrm{Reset}(\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3\mathsf{n}_4)$
$= \mathscr{B}_{s_2}\langle 1,\mathsf{n}_4\mathsf{n}_1,\mathsf{n}_2\rangle$	1	$s_1 \neq s_2$
$= \max(\mathscr{B}_{s_2}\langle 2,\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2,\mathsf{n}_3\rangle,\mathscr{B}_{s_2}\langle 2,\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2,\mathsf{n}_5\rangle)$	2	$\neg\mathrm{Reset}(\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2),\neg\mathrm{Reset}(\mathsf{n}_4\mathsf{n}_1\mathsf{n}_3)$
$= \max(\mathscr{B}_{s_2}\langle 2,\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3,\mathsf{n}_4\rangle,\mathscr{B}_{s_2}\langle 2,\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2\mathsf{n}_5,\mathsf{end}\rangle)$	2	$s_3 \neq s_2$
$= \max(\mathscr{B}_{s_2}\langle 1,\mathsf{n}_4,\mathsf{n}_1\rangle,0)$	**2**	$\mathrm{Reset}(\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3\mathsf{n}_4)$

The algorithm starts with n_1, the root of G. Since n_1 uses buffer s_1 (different from s_2), we continue with the successor n_2. It uses s_2 and, since the accumulated path $\mathsf{n}_1\mathsf{n}_2$ does not satisfy the reset property, the buffer requirement of s_2 needs to be increased to 1. The next nodes, n_3 and n_5, do not use the channel s_2. Since n_4 uses s_2 and $\mathrm{Reset}(\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3\mathsf{n}_4)$ holds (there is $\mathsf{n}_2 \prec_{\mathtt{IO}} \mathsf{n}_3 \prec_{\mathtt{IO}} \mathsf{n}_4$), the buffer has to be emptied before n_4: we thus reinitialise the buffer requirement to 1 and the path to just n_4. On the other branch, we reach end and stop the computation. The next prefix of n_4, n_1, does not use s_2, but it successor n_2 does. We thus check the reset property on the path $\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2$, but it does not hold. The buffer requirement is thus increased to 2. As previously, n_3 and n_5 do not use the channel s_2 and the accumulated path (in the main branch) becomes $\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3$. The next prefix, n_4, uses s_2 and $\mathrm{Reset}(\mathsf{n}_4\mathsf{n}_1\mathsf{n}_2\mathsf{n}_3\mathsf{n}_4)$ holds: thus we initialise the buffer requirement back to 1 and the path to just n_4. However, we just explored such a situation earlier in the computation and thus stop. The maximum buffer size encountered for s_2 is then 2. Such a computation for s_1 and s_3 gives a buffer size of 1.

3.3 Subject Reduction and Buffer Safety

Once global type G is agreed upon by all parties, a local type T_i from each party's viewpoint is generated as a projection of G, and implemented as a process P_i. If all the resulting local processes $P_1,..,P_n$ can be type-checked against $T_1,..,T_n$, they are automatically guaranteed to interact properly, without communication mismatch (communication safety) nor getting stuck inside a session (progress) [11]. Here we additionally ensure the absence of buffer-overflow based on the buffer bound analysis of G.

Local types. Local session types type-abstract sessions from each end-point's view.

$$T \ ::= \ k!\langle U\rangle;T \ \mid \ k?\langle U\rangle;T \ \mid \ k\oplus\{l_i\colon T_i\}_{i\in I} \ \mid \ k\&\{l_i\colon T_i\}_{i\in I} \ \mid \ \mu\mathbf{x}.T \ \mid \ \mathbf{x} \ \mid \ \mathsf{end}$$

Type $k!\langle U\rangle$ expresses the sending to k of a value of type U. Type $k?\langle U\rangle$ is its dual. Type $k\oplus\{l_i\colon T_i\}_{i\in I}$ represents the transmission to k of a label l_i chosen in the set $\{l_i \mid i\in I\}$, followed by the communications described by T_i. Type $k\&\{l_i\colon T_i\}_{i\in I}$ is its dual. The remaining type constructors are standard. We say a type is *guarded* if it is neither a recursive type nor a type variable. The relation between global and local types is formalised by *projection*, written $G\upharpoonright \mathtt{p}$ (called *projection of* G *onto* $\mathtt{p}$) and defined in [11,22]. For example, $(\mathtt{p}\to\mathtt{p}'\colon k\,\langle U\rangle;G')\upharpoonright\mathtt{p} = k!\,\langle U\rangle;(G'\upharpoonright\mathtt{p})$, $(\mathtt{p}\to\mathtt{p}'\colon k\,\langle U\rangle;G')\upharpoonright\mathtt{p}' = k?\,\langle U\rangle;(G'\upharpoonright\mathtt{p}')$ and $(\mathtt{p}\to\mathtt{p}'\colon k\,\langle U\rangle;G')\upharpoonright\mathtt{q} = (G'\upharpoonright\mathtt{q})$. We take an *equi-recursive* view, not distinguishing between $\mu\mathbf{x}.T$ and its unfolding $T[\mu\mathbf{x}.T/\mathbf{x}]$.

Linearity. To avoid race conditions and conflicts between typed processes, we build on the definition of linearity from [11]. The relations $\prec_{\mathtt{II}}$ and $\prec_{\mathtt{OO}}$ are defined by:

$$\mathsf{n}_1\prec_{\mathtt{II}}\mathsf{n}_2\ \text{ if } \mathsf{n}_1\prec\!\!\prec\mathsf{n}_2 \text{ and } \mathtt{pfx}(\mathsf{n}_1)=\mathtt{p}_1\to\mathtt{p}:k_1 \text{ and } \mathtt{pfx}(\mathsf{n}_2)=\mathtt{p}_2\to\mathtt{p}:k_2 \text{ s.t. } \mathtt{p}_1\neq\mathtt{p}_2\Leftrightarrow k_1\neq k_2$$

$$\mathsf{n}_1\prec_{\mathtt{OO}}\mathsf{n}_2\ \text{ if } \mathsf{n}_1\prec\!\!\prec\mathsf{n}_2 \text{ and } \mathtt{pfx}(\mathsf{n}_1)=\mathtt{p}\to\mathtt{p}_1:k_1 \text{ and } \mathtt{pfx}(\mathsf{n}_2)=\mathtt{p}\to\mathtt{p}_2:k_2 \text{ s.t. } \mathtt{p}_1\neq\mathtt{p}_2\Rightarrow k_1\neq k_2$$

The three relations $\prec_{\mathtt{IO}}$, $\prec_{\mathtt{II}}$ and $\prec_{\mathtt{OO}}$ are used to characterise the authorised sequences of actions. An *input dependency (I-dependency) from* n_1 *to* n_2 is a chain $\mathsf{n}_1\prec_{\phi_1}\cdots\prec_{\phi_n}\mathsf{n}_2$ $(n\geq 1)$ such that $\phi_i=\mathsf{IO}$ for $1\leq i\leq n-1$ and $\phi_n=\mathsf{II}$. An *output dependency (O-dependency) from* n_1 *to* n_2 is a chain $\mathsf{n}_1\prec_{\phi_1}\cdots\prec_{\phi_n}\mathsf{n}_2$ $(n\geq 1)$ such that $\phi_i\in\{\mathsf{OO},\mathsf{IO}\}$. These dependency relations are respectively written $\prec\!\!\prec_{\mathtt{II}}$ and $\prec\!\!\prec_{\mathtt{OO}}$. G is *linear* (written $\mathrm{Lin}(G)$) if, whenever two nodes $\mathsf{n}_1\prec\!\!\prec\mathsf{n}_2$ use the same channel k, the dependencies $\mathsf{n}_1\prec\!\!\prec_{\mathtt{II}}\mathsf{n}_2$ and $\mathsf{n}_1\prec\!\!\prec_{\mathtt{OO}}\mathsf{n}_2$ hold. If G carries other global types, we inductively demand the same. Examples can be found in [5,11]. We call linear global types whose projections are defined *coherent*. Hereafter we only consider coherent types.

Typing initial processes. The type judgements for initial processes are of the form $\Gamma\vdash P\triangleright\Delta$ which means: "under the environment Γ, process P has typing Δ". Environments are defined by: $\Gamma\ ::=\ \emptyset\mid\Gamma,u:S\mid\Gamma,X:\Delta$ and $\Delta\ ::=\ \emptyset\mid\Delta,\tilde{s}^{\tilde{m}}:\{T\,@\mathtt{p}\}_{\mathtt{p}\in I}$. A *sorting* $(\Gamma,\Gamma',..)$ is a finite map from names to sorts and from process variables to sequences of sorts and types. *Typing* $(\Delta,\Delta',..)$ records linear usage of session channels. In multiparty sessions, it assigns a family of located types to a vector of session channels. In addition, we annotate each session channel s_k with its buffer bound m_k.

Among the typing rules, the rule for session initiation uses the buffer size $\mathscr{B}_{s_i}\langle G\rangle$ calculated from G.

$$\frac{\Gamma\vdash a:G\quad \Gamma\vdash P\triangleright\Delta,\tilde{s}^{\tilde{m}}:(G\upharpoonright 1)@\mathtt{1}\ \ |\tilde{s}|=\mathit{chans}(G)\ \ \mathscr{B}_k\langle G\rangle=m_k}{\Gamma\vdash\overline{a}_{[\mathtt{2..n}]}(\tilde{s}^{\tilde{m}}).P\triangleright\Delta}$$

The type for $\tilde{s}$ is the *first* projection of the declared global type for a in Γ. The end-point type $(G\upharpoonright\mathtt{p})@\mathtt{p}$ means that the participant $\mathtt{p}$ has $G\upharpoonright\mathtt{p}$, which is the projection of G onto $\mathtt{p}$, as its end-point type. The condition $|\tilde{s}|=\mathit{chans}(G)$ means the number of session channels meets those in G. The condition $\mathscr{B}_k\langle G\rangle=m_k$ ensures that the size of the buffer m_i for each s_k does not exceed the size calculated from G. Similarly for accept. Other rules for initial processes are identical with [11]. Note that since $\mathscr{B}_k\langle G\rangle$ is decidable, type-checking for processes with type annotations is decidable [11,22].

The rest of the typing system for programs and one for runtime are similar with those in [11] ([5]). Judgements for runtime are there extended to $\Gamma\vdash_\Sigma P\triangleright\Delta$ with Σ a set of session channels associated to the current queue.

For the subject reduction, we need to keep track of the correspondence between the session environment and the buffer sizes. We use the reduction over session typing, $\Delta \xrightarrow{k} \Delta'$, that is generated by rules between types such as $k!\langle U\rangle;T\,@\mathtt{p}, k?\langle U\rangle;T'\,@\mathtt{q} \xrightarrow{k} T\,@\mathtt{p}, T'\,@\mathtt{q}$. The key lemma about the correspondence between buffer size and reduction follows. We set $[\![G]\!]$ to be the family $\{(G\upharpoonright \mathtt{p})@\mathtt{p} \mid \mathtt{p} \in G\}$. Regarding each type in $[\![G]\!]$ as the corresponding regular tree, we can define $\prec$, $\prec_{\mathtt{II}}, \prec_{\mathtt{IO}}$ and $\prec_{\mathtt{OO}}$ among its prefixes precisely as we have done for G.

Lemma 3.7 *If $\Delta(\tilde{s}) = [\![G]\!]$ and $\Delta \xrightarrow{s_k} \Delta'$, then $[\![G]\!](\xrightarrow{k})^*[\![G']\!]$ with $\Delta'(\tilde{s}) = [\![G']\!]$ and $\mathscr{B}_k\langle G\rangle \geq \mathscr{B}_k\langle G'\rangle$.*

When $\Gamma \vdash_\Sigma P \triangleright \Delta$, we say that $(\Gamma, \Sigma, P, \Delta)$ is *fully coherent* for session $\tilde{s}$ if there exist $P_1, \ldots, P_k, \Sigma', \Delta'$ such that $\Gamma \vdash_{\Sigma \uplus \Sigma'} P \mid P_1 \mid \ldots \mid P_k \triangleright \Delta, \Delta'$ and $\Delta, \Delta' = \Delta'', \tilde{s}^{\tilde{n}} : \{T_\mathtt{p}@\mathtt{p}\}_{\mathtt{p}\in I}$ with $[\![G]\!] = \{T_\mathtt{p}@\mathtt{p}\}_{\mathtt{p}\in I}$, G coherent and $\mathscr{B}_i\langle G\rangle \leq n_i$ $(1 \leq i \leq k)$.

Theorem 3.8 (Subject Reduction) *$\Gamma \vdash_\Sigma P \triangleright \Delta$ and $P \longrightarrow Q$ with $(\Gamma, \Sigma, P, \Delta)$ fully coherent imply $\Gamma \vdash_\Sigma Q \triangleright \Delta'$ for some Δ', s_k such that $\Delta = \Delta'$ or $\Delta(\xrightarrow{s_k})^*\Delta'$ and $\mathscr{B}_k\langle G\rangle \geq \mathscr{B}_k\langle G'\rangle$ with $\Delta(\tilde{s}) = [\![G]\!]$, $\Delta'(\tilde{s}) = [\![G']\!]$ and $(\Gamma, \Sigma, Q, \Delta')$ fully coherent.*

The proof relies on Lemma 3.7 and the fact that session reduction does not affect the causal dependencies within global types, so that buffer sizes can only decrease.

To state our buffer safety result, we define the *buffer overflow error* as follows:

$$\begin{array}{lcl} n \leq |\tilde{h}| & \Rightarrow & s!\langle\tilde{e}\rangle;P \mid s^n:\tilde{h} \rightarrow \mathsf{Err},\ s!\langle\!\langle\tilde{t}\rangle\!\rangle;P \mid s^n:\tilde{h} \rightarrow \mathsf{Err},\ s \lhd l;P \mid s^n:\tilde{h} \rightarrow \mathsf{Err} \\ P \rightarrow \mathsf{Err} & \Rightarrow & P \mid Q \rightarrow \mathsf{Err},\ (\nu\ a)P \rightarrow \mathsf{Err},\ (\nu\ \tilde{s})P \rightarrow \mathsf{Err},\ P \equiv Q \rightarrow \mathsf{Err} \end{array}$$

Corollary 3.9 (Buffer Safety) *If $\Gamma \vdash_\Sigma P \triangleright \Delta$, then for all P' s.t. $P \longrightarrow^* P'$, $P' \not\rightarrow \mathsf{Err}$.*

4 Channel Attribution

This section describes algorithms that attribute channels to the communications of a given global type without channels, called *stripped global types* $(\underline{G}, \underline{G}', \ldots)$ defined as:

$$\underline{G} ::= \ \ldots \mid \mathtt{p} \rightarrow \mathtt{p}'\langle U\rangle;\underline{G}' \mid \mathtt{p} \rightarrow \mathtt{p}'\{l_j : \underline{G}_j\}_{j\in J} \qquad \text{values, branching}$$

Our algorithms transform $\underline{G}$ into regular type G by adding channel annotations. We define the *channel allocation* of a global type G to be the value of the function $\mathtt{ch}$.

Singleton allocation. The simplest channel allocation attributes a different channel to each communication occurring in the global type syntax tree. Formally, the singleton allocation is such that: $\forall \mathsf{n}, \mathsf{n}' \in G,\ \mathtt{ch}(\mathsf{n}) = \mathtt{ch}(\mathsf{n}') \iff \mathsf{n} = \mathsf{n}'$. Singleton allocations enjoy the following good properties.

Lemma 4.1 *For any global type G with singleton allocation, (1) G satisfies the linearity property; (2) for the finite channels k of G, $\mathscr{B}_k\langle G\rangle = 1$; (3) for the finite channels k of G, $\sum_k \mathscr{B}_k\langle G\rangle \leq size(G)$.*

Channel equalities. As well as values of the $\mathtt{ch}$ function, allocations can be seen as partitions of the set of nodes $\{\mathsf{n}\}_{\mathsf{n}\in G}$. We then define partition refinement through the notion of *channel equality*, i.e. the union of two partitions to produce a new allocation.

Definition 4.2 (channel equality) A *channel equality* is the substitution of two channels k and k' in a global type G by a single fresh channel k'' while keeping G linear.

As we take the singleton allocation as a base, we can describe channel allocations by sets E of channel equalities, the empty set corresponding to the singleton allocation. We write G_E the global type G with channel equalities E.

In the rest of this section, we always start from the singleton allocation and proceed by channel equality. We notably rely on the fact that the result of the equality of two finite channels is finite. Formally, if $\mathscr{B}_k\langle G\rangle = \infty$ then $\forall E, \mathscr{B}_k\langle G_E\rangle = \infty$.

Note that the total number of possible channel allocations is finite and corresponds to the number of partitions of a given finite set. The exact count (if we do not take into account the linearity property) is given by a Bell number [19] which is exponential in the size of the global type. Given the finite number of possible allocations, we know that there exists an algorithm to find allocations satisfying any decidable property. Notably, one can reach any given memory requirement (number of channels, buffer sizes).

Principal allocation. The most widely used allocation method attributes two communication channels (one in each direction) for each pair of participants. The session types in [1, 8] follow this allocation. Formally, the principal allocation is such that: $\forall \mathsf{n},\mathsf{n}' \in G$ s.t. $\mathtt{pfx}(\mathsf{n}) = \mathtt{p} \to \mathtt{q} : k$ and $\mathtt{pfx}(\mathsf{n}') = \mathtt{p}' \to \mathtt{q}' : k', (k = k' \iff \mathtt{p} = \mathtt{p}' \wedge \mathtt{q} = \mathtt{q}')$.

Lemma 4.3 *For any global type G with principal allocation, (1) G satisfies the linearity property; (2) chans$(G) \leq n \times (n-1)$ where $n =$ prins(G).*

Greedy allocations. We now define a family of efficient algorithms, that give good allocation results in practice.

Definition 4.4 (Greedy allocation algorithm) *Given a global type with singleton allocation G, of initial node* n_0*, and a successor function* $\mathtt{succ}$ *over the nodes, the function* $\mathscr{I}_{\emptyset}^{\emptyset}(\mathsf{n}_0)$ *is defined by:*

$$\begin{array}{l} \mathscr{I}_E^{\mathrm{K}}(\mathsf{n}) = \mathscr{I}_{E'}^{\mathrm{K}'}(\mathsf{n}') \textit{ where } \begin{cases} \mathtt{succ}(\mathsf{n}) = \mathsf{n}' \\ \mathtt{ch}(\mathsf{n}) = k \\ \mathrm{K}' = \mathrm{K} \cup \{k\} \end{cases} \wedge E' = \begin{cases} E \cup \{k = k'\} & \textit{if } \exists k' \in \mathrm{K}, \mathrm{Lin}(G_{E\cup\{k=k'\}}) \\ E & \textit{otherwise} \end{cases} \\ \mathscr{I}_E^{\mathrm{K}}(\mathit{end}) = E \end{array}$$

This algorithm is parameterised by the successor function over nodes (that can be given e.g. by a depth-first graph search) and by the choice between the possible channels $k' \in \mathrm{K}$ for equality. The greedy algorithm has the advantage of not backtracking and thus enjoys a polynomial complexity (if the choice procedures are polynomial) in the size of the graph. In particular, we define two efficient heuristics based on the generic greedy algorithm. In the greedy algorithm, we implement K by either:

1. (Early) a queue, so that we choose for channel equality the oldest channel $k' \in \mathrm{K}$.
2. (Late) a list, so that we choose for channel equality the latest channel $k' \in \mathrm{K}$.

The early and late allocations are not optimal in terms of total memory requirements (computed by $\sum_k \mathscr{B}_k\langle G\rangle$ when all channels are finite) but give good results in practice while being polynomial.

Example 4.5 (comparison of the allocations)
We apply the different allocation algorithms on a three-party stripped global type. The results are given in the adjacent table in term of number of allocated channels and total memory requirement. The greedy algorithms give the best results on this example, with the early greedy algorithm allocating less channels than the late greedy algorithm.

	Singleton	Principal	Early G.	Late G.
$\mathsf{n}_0 : \mathtt{A} \to \mathtt{B};$	k_0	k_0	k_0	k_0
$\mathsf{n}_1 : \mathtt{B} \to \mathtt{A};$	k_1	k_1	k_1	k_1
$\mathsf{n}_2 : \mathtt{A} \to \mathtt{B};$	k_2	k_0	k_0	k_0
$\mathsf{n}_3 : \mathtt{A} \to \mathtt{B};$	k_3	k_0	k_0	k_1
$\mathsf{n}_4 : \mathtt{A} \to \mathtt{C};$	k_4	k_2	k_2	k_2
$\mathsf{n}_5 : \mathtt{C} \to \mathtt{B};$	k_5	k_3	k_1	k_3
$\mathsf{n}_6 : \mathtt{B} \to \mathtt{C};$	k_6	k_4	k_2	k_1
$\mathsf{n}_7 : \mathtt{B} \to \mathtt{C}$	k_7	k_4	k_2	k_2
Nb channels	8	5	3	4
Memory Req.	8	7	5	5

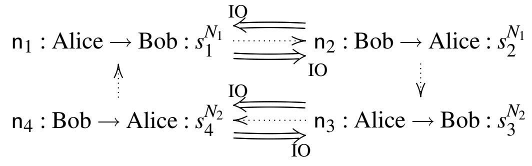
5 Local Refinement: Messaging Optimisations

One of the significant practical concerns in systems with messaging is to optimise interactions through more asynchronous data processing to increase parallelism. Our recent work [14, 15] developed a new form of subtyping, the *asynchronous subtyping*, that characterises the compatibility between classes of type-safe permutations of actions, in order to send messages before receiving. This subtyping allows, however, not only Bob_5 in (1.1) in § 1 but also $\mu X.s_2!\langle \mathit{Orange}\rangle;X$ as a refinement of Bob, which changes all buffer sizes from 1 to ∞, leading to buffer overflows. Our aim is to overcome this problem by controlling permutations *locally* with the help of the IO-dependency analysis. The key idea is to prohibit the permutation of an output action at k_0 with an input or branching action which prevents (by IO-causality) the accumulation of values in k_0.

Recall Definition 3.4. We define the *minimal resetting paths* to be the paths that satisfy the reset property while none of their suffix does. Then, we define the *dependent nodes* of channel k, noted $\mathtt{dep}(k)$ to be the union of the reset nodes of the minimal resetting paths that end with k. This set of nodes characterises a buffer usage.

First, for a given G, we choose a partition $\{N_0,\ldots,N_n\}$ of the set of nodes of G. This partition should satisfy the two properties: $\forall \mathsf{n} \in N_i, \mathsf{n}' \in N_j, \mathtt{ch}(\mathsf{n}) = \mathtt{ch}(\mathsf{n}') \Rightarrow N_i = N_j$ and $\forall \mathsf{n} \in N_i, \mathtt{dep}(\mathtt{ch}(\mathsf{n})) \subset N_i$. The choice of a partitioning depends in particular on a choice of reset and dependent nodes. Note that the trivial partitioning (with only one partition) is always possible. Since that, for each channel k, all nodes using k are part of the same partition (written $N(k)$), we can annotate all uses of k in G by $N(k)$.

In the example below, the partitioning is made of $N_1 = \{\mathsf{n}_1,\mathsf{n}_2\}$ and $N_2 = \{\mathsf{n}_3,\mathsf{n}_4\}$: we give the annotated session graph (with the IO-dependencies highlighted) on the left and the projected types (where the annotations are kept) on the right.


$\mathsf{n}_1 : \mathrm{Alice} \to \mathrm{Bob} : s_1^{N_1}$
$\mathsf{n}_2 : \mathrm{Bob} \to \mathrm{Alice} : s_2^{N_1}$
$\mathsf{n}_4 : \mathrm{Bob} \to \mathrm{Alice} : s_4^{N_2}$
$\mathsf{n}_3 : \mathrm{Alice} \to \mathrm{Bob} : s_3^{N_2}$
IO
IO
IO
IO

$$T_{\mathrm{Alice}} = \mu\mathbf{x}.s_1^{N_1}!;s_2^{N_1}?;s_3^{N_2}!;s_4^{N_2}?;\mathbf{x}$$
$$T_{\mathrm{Bob}} = \mu\mathbf{x}.s_1^{N_1}?;s_2^{N_1}!;s_3^{N_2}?;s_4^{N_2}!;\mathbf{x}$$
$$T^{opt}_{\mathrm{Alice}} = \mu\mathbf{x}.s_1^{N_1}!;s_3^{N_2}!;s_2^{N_1}?;s_4^{N_2}?;\mathbf{x}$$

Next, we apply the size-preserving asynchronous communication subtyping, following the annotations on the projected types. The relation $T \ll T'$ means T is more asynchronous than (or more optimised than) T'. The main rule is:

$$(\mathsf{OI}) \qquad k^{N}!\langle U\rangle;k_0^{N_0}?\langle U'\rangle;T \ll k_0^{N_0}?\langle U'\rangle;k^{N}!\langle U\rangle;T \qquad (N\cap N_0=\emptyset)$$

where the two prefixes are permutable if the IO-chains of the two prefixes are disjoint. We can *always* permute two inputs and two outputs with distinct channels since they do not contribute to the input and output alternations that constitute the IO-chains. The branching/selection rules are similarly defined, and others are context rules. Then we define a coinductive subtyping relation $T_1 \leqslant_c T_2$ as a form of type simulation, following [14, 15]. The important fact is that $\leqslant_c$ does not alter buffer sizes: suppose $[\![G]\!] = \{T\,@\mathtt{p}\}_{\mathtt{p}}$ with $T\,@\mathtt{p} = (G\upharpoonright\mathtt{p})@\mathtt{p}$ and $\mathtt{p} \in G$. Assume $T\,@\mathtt{p} \leqslant_c T'\,@\mathtt{p}$ with $[\![G']\!] = \{T'\,@\mathtt{p}\}_{\mathtt{p}}$ Then $\mathscr{B}_k\langle G\rangle = \mathscr{B}_k\langle G'\rangle$. Since there is no change in the buffer bounds, Type and Buffer Safety are just proved from Theorem 3.8 and Corollary 3.9.

In the example above, in Alice's type, we can permute $s_2^{N_1}?$ and $s_3^{N_2}!$ ($T_{\text{Alice}}^{opt} \leqslant_c T_{\text{Alice}}$) since $N_1 \cap N_2 = \emptyset$, keeping the size of each buffer one. Hence process typable by T_{Alice}^{opt} can safely send message at s_3 before input at s_2. In Alice-Bob$_5$-Carol from § 1, the original global type G annotated by IO-chains has only one partition $N = \{\mathsf{n}_1, \mathsf{n}_2, \mathsf{n}_3\}$:

$$\mu\mathbf{x}.\ \text{Alice} \to \text{Bob}:\ s_1^N\,\langle \mathsf{nat}\rangle;\text{Bob} \to \text{Carol}:\ s_2^N\,\langle \mathsf{string}\rangle;\text{Carol} \to \text{Alice}:\ s_3^N\,\langle \mathsf{real}\rangle;\mathbf{x}$$

Bob's local type is $\mu\mathbf{x}.s_1^N?\langle\mathsf{nat}\rangle;s_2^N!\langle\mathsf{string}\rangle;\mathbf{x}$, which prevents any optimisation $\ll$ by (OI). Hence, Bob$_5$ is not typable. Some typable examples are given in the next section.

5.1 Application: Multi-buffering algorithm

The double buffering algorithm [6] is widely used in high-performance and multicore computing. We generalise this algorithm to *multi-buffering* [16], and solve an open issue in our previous work [15, § 5]. The aim is to transport a large amount of data as a series of units (say each unit is 16kB) from a source (Source) to a transformer (called Kernel). Each unit gets processed at Kernel and delivered to a sink (Sink). Kernel uses n 16kB buffers, named B_i, to maximise the message transfer asynchrony. Processes which represent Source, Sink, Kernel and Optimised Kernel are given below using parameterised processes [22] (i.e. where $\mathtt{foreach}(i<n)\{P[i]\}$ means we iterate $P[i]$ for $0 \leq i < n$):

Source: $\mu X.\mathtt{foreach}(i<n)\{r_i?();\ s_i!\langle y_i\rangle\};X$ *Sink:* $\mu X.\mathtt{foreach}(i<n)\{t_i!\langle\rangle;u_i?(z_i)\};X$
Kernel: $\mu X.\mathtt{foreach}(i<n)\{r_i!\langle\rangle;s_i?(x_i);t_i?();u_i!\langle x_i\rangle\};X$
Optimised Kernel: $r_0!\langle\rangle;\ldots;r_{n-1}!\langle\rangle;\mu X.\mathtt{foreach}(i<n)\{s_i?(x_i);t_i?();u_i!\langle x_i\rangle;r_i!\langle\rangle\};X$

In the loop, Kernel notifies Source with signals at r_i that it is ready to receive data in each channel s_i of buffer B_i. Source complies, sending one unit via s_i. Then Kernel waits for Sink to inform (via t_i) that Sink is ready to receive data via u_i: upon receiving the signals, Kernel sends the unit of processed data to Sink via u_i. If Kernel sends the n notifications to $r_0,\ldots,r_{n-1}$ *ahead* like Optimised Kernel, Source can start its work for the next unit (sending y_j at s_j) without waiting for other buffers.

The following proposition means that *the n-buffers of size one in Kernel simulate one buffer of size n*, maximising the asynchrony. The proof is done by annotating the global type with partitions $\{r_i, s_i\}$ and $\{u_i, t_i\}$, and checking that the permutation of the projected Kernel type satisfies the (OI) rule.

Proposition 5.1 (**n-buffering correctness**) *Source-Optimal Kernel-Sink satisfies both progress and communication-safety. Also each buffer at s_i and u_i holds at most one unit.*

If Optimised Kernel is optimised as $r_0!\langle\rangle;...;\mathtt{foreach}(i < n)\{r_i!\langle\rangle;s_i?(x_i);t_i?;u_i!\langle x_i\rangle\}$ (which is not typable in our system), then all buffers are forced to hold 2 *units*. This unsafe optimisation is typable in [15] but prevented here by Proposition 5.1. In [5], we also deal with a use case of MPSoC buffer allocation from [4], with branching and iterations [22], and verify it by applying all of the previously described methods.

6 Related Work

Checking buffer bounds based on global specifications has been studied through Petri nets and Synchronous data flow. Recent advances [9] in the study of Kahn Process Networks (KPN) have improved Parks's algorithm [18] to ensure safe executions of stream-based applications with bounded buffers, using an appropriate scheduling policy. Their theory is applied to KPN applications on MPSoC [4], demonstrating the effectiveness of non-uniform, fine-grained buffer allocations. By contrast, our approach is type-based and relies on the existence of a global specification that brings additional guarantees (such as deadlock-freedom) and allows global choreography manipulation and refinements. It is moreover directly applicable to programming languages [12, 22] by extending existing type syntax and checking.

The idea of using a type-abstraction to investigate channel communications goes back to Nielson & Nielson's work on CML [17]. Gay & Vasconcelos [8] propose a linear type system for binary sessions to enforce buffer bounds computed by a fixed point method. Their work is thus limited to a particular channel allocation (i.e. principal, cf. § 4) and does not extend to multiparty interactions (their method would find that the buffers in Example (a) are infinite). Terauchi & Megacz [21] describe a polynomial method to infer buffer bounds of a concurrent language through program analysis using linear programming techniques, improving on previous work in [13], see [21, § 7]. Our bound computation method differs in that it starts from a direct type-based abstraction of global interaction structures, namely session graphs, not from direct investigation of local types nor processes (normally in distributed systems, a peer does not know other peer's type or implementation [12]). It also leads to the general simplicity of the analysis, and the uniform treatment of subtle issues such as asynchronous optimisations. Thanks to session types, the channel passing problem in [21, § 6] does not arise in our analysis: different (possibly newly generated) sessions and names can be stored in the same buffer, still giving the exact bound of stored channels. None of [8, 21] have studied either channel allocation, global refinement or messaging optimisation.

Among process calculi for service-oriented computing (SOC), contracts [3] and the conversation calculus [2] provide static type checking for a series of interactions and ensure progress. We demonstrate the advantage of global types by the simplicity of our analysis and the uniform treatments and articulation of our various algorithms. Our approach is, however, extensible to these calculi because (1) the IO-causality analysis does not rely on the form of session branches so that other form of sums can be analysed by the same technique; and (2) combining with a polynomial inference which builds a graph from a collection of local types $[\![G]\!]$ [15], Subject Reduction Theorem can be proved using our invariance method noting that we use $[\![G]\!]$ for the proofs. An extension to other formalisms for SOC including [2, 3] is an interesting future work.

Further topics include the enrichment of global types with more quantitative information (such as distance, probabilities and weights), which would enable finer-grained analyses and optimisations.

References

1. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433, 2008.
2. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300. Springer, 2009.
3. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR*, number 5710 in LNCS, pages 211–228, 2009.
4. E. Cheung, H. Hsieh, and F. Balarin. Automatic buffer sizing for rate-constrained KPN applications on multiprocessor system-on-chip. In *HLDVT'07*, pages 37–44. IEEE, 2007.
5. P.-M. Deniélou and N. Yoshida. Buffered communication analysis in distributed multiparty sessions. Full version, Prototype at `http://www.doc.ic.ac.uk/~pmalo/multianalysis`.
6. A. Donaldson, D. Kroening, and P. Rmmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, LNCS 6015, pages 280–295, 2010.
7. M. Fähndrich et al. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
8. S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 2009.
9. M. Geilen and T. Basten. Requirements on the Execution of Kahn Process Networks. In *ESOP*, volume 2618 of *LNCS*, pages 319–334. Springer, 2003.
10. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
11. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284, 2008.
12. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, 2008.
13. N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *SAS*, LNCS 983, pages 225–242, 1995.
14. D. Mostrous and N. Yoshida. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In *TLCA*, volume 5608 of *LNCS*, pages 203–218. Springer, 2009.
15. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP'09*, volume 5502 of *LNCS*, pages 316–332, 2009.
16. Multi-buffering. `http://en.wikipedia.org/wiki/Multiple_buffering`.
17. H. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *POPL*, pages 84–97, 1994.
18. T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, California Barkeley, 1995.
19. G.-C. Rota. The number of partitions of a set. *Amer. Math. Monthly*, 71:498–504, 1964.
20. K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer, 1994.
21. T. Terauchi and A. Megacz. Inferring channel buffer bounds via linear programming. In *ESOP*, volume 4960 of *LNCS*, pages 284–298. Springer, 2008.
22. N. Yoshida, P.-M. Deniélou, A. Bejleri, and R. Hu. Parameterised multiparty session types. In *FoSSaCs*, volume 6014 of *LNCS*, pages 128–145, 2010.