Deadlock Freedom for Asynchronous and Cyclic Process Networks*

Bas van den Heuvel and Jorge A. Pérez
University of Groningen, The Netherlands

This paper considers the challenging problem of establishing deadlock freedom for message-passing processes using behavioral type systems. In particular, we consider the case of processes that implement session types by communicating asynchronously in cyclic process networks. We present APCP, a typed process framework for deadlock freedom which supports asynchronous communication, delegation, recursion, and a general form of process composition that enables specifying cyclic process networks. We discuss the main decisions involved in the design of APCP and illustrate its expressiveness and flexibility using several examples.

1 Introduction

Modern software systems often comprise independent components that interact by passing messages. The π-calculus is a consolidated formalism for specifying and reasoning about message-passing processes [19, 20]. Type systems for the π-calculus can statically enforce communication correctness. In this context, *session types* are a well-known approach, describing two-party communication protocols for channel endpoints and enforcing properties such as *protocol fidelity* and *deadlock freedom*.

Session type research has gained a considerable impulse after the discovery by Caires and Pfenning [7] and Wadler [28] of Curry-Howard correspondences between session types and linear logic [12]. Processes typable in type systems derived from these correspondences are inherently deadlock free. This is because the CUT-rule of linear logic imposes that processes in parallel must connect on exactly one pair of dual endpoints. However, whole classes of deadlock free processes are not expressible with the restricted parallel composition and endpoint connection resulting from CUT [9]. Such classes comprise *cyclic process networks* in which parallel components are connected on multiple endpoints at once. Defining a type system for deadlock free, cyclic processes is challenging, because such processes may contain *cyclic dependencies*, where components are stuck waiting for each other.

Advanced type systems that enforce deadlock freedom of cyclic process networks are due to Kobayashi [17], who exploits *priority annotations* on types to avoid circular dependencies. Dardha and Gay bring these insights to the realm of session type systems based on linear logic by defining Priority-based CP (PCP) [8]. Indeed, PCP incorporates the type annotations of Padovani's simplification of Kobayashi's type system [21] into Wadler's Classical Processes (CP) derived from classical linear logic [28].

In this paper, we study the effects of *asynchronous communication* on type systems for deadlock free cyclic process networks. To this end, we define *Asynchronous PCP* (APCP), which combines Dardha and Gay's type annotations with DeYoung *et al.*'s semantics for asynchronous communication [10], and adds support for tail recursion. APCP uncovers fundamental properties of type systems for asynchronous communication, and simplifies PCP's type annotations while preserving deadlock freedom results.

*Research partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

J. Lange, A. Mavridou, L. Safina, A. Scalas (Eds.):
14th Interaction and Concurrency Experience (ICE 2021).
EPTCS 347, 2021, pp. 38–56, doi:10.4204/EPTCS.347.3

© Van den Heuvel & Pérez
This work is licensed under the
Creative Commons Attribution License.

In Section 2, we motivate APCP by discussing Milner's cyclic scheduler [19]. Section 3 defines APCP's language and type system, and proves Type Preservation (Theorem 2) and Deadlock Freedom (Theorem 6). In Section 4, we showcase APCP by returning to Milner's cyclic scheduler and using examples inspired by Padovani [21] to illustrate asynchronous communication and deadlock detection. Section 5 discusses related work and draws conclusions.

2 Motivating Example: Milner's Cyclic Scheduler

We motivate by example the development of APCP, our type system for deadlock freedom in asynchronous, cyclic message-passing processes. We consider Milner's cyclic scheduler [19], which crucially relies on asynchrony and recursion. This example is inspired by Dardha and Gay [8], who use PCP to type a synchronous, non-recursive version of the scheduler.

The system consists of $n \geq 1$ worker processes P_i (the workers, for short), each attached to a partial scheduler A_i. The partial schedulers connect to each other in a ring structure, together forming the cyclic scheduler. The scheduler then lets the workers perform their tasks in rounds, each new round triggered by the leading partial scheduler A_1 (the *leader*) once each worker finishes their previous task. We refer to the non-leading partial schedulers A_{i+1} for $1 \leq i < n$ as the *followers*.

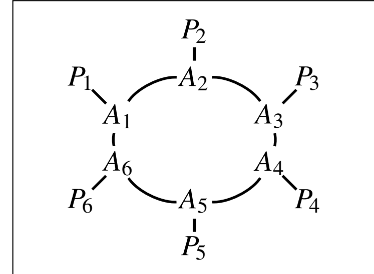

Figure 1: Milner's cyclic scheduler with 6 workers.
Lines denote channels connecting processes.

Each partial scheduler A_i has a channel endpoint a_i to connect with the worker P_i's channel endpoint b_i. The leader A_1 has an endpoint c_n to connect with A_n and an endpoint d_1 to connect with A_2 (or with A_1 if $n = 1$; we further elude this case for brevity). Each follower A_{i+1} has an endpoint c_i to connect with A_i and an endpoint d_{i+1} to connect with A_{i+2} (or with A_1 if $i+1 = n$; we also elude this case).

In each round of the scheduler, each follower A_{i+1} awaits a start signal from A_i, and then asynchronously signals P_{i+1} and A_{i+2} to start. After awaiting acknowledgment from P_{i+1} and a next round signal from A_i, the follower then signals next round to A_{i+2}. The leader A_1, responsible for starting each round of tasks, signals A_2 and P_1 to start, and, after awaiting acknowledgment from P_1, signals next round to A_2. Then, the leader awaits A_n's start and next round signals.

It is crucial that A_1 does not await A_n's start signal before starting P_1, as the leader would otherwise not be able to initiate rounds of tasks. Asynchrony thus plays a central role here: because A_n's start signal is non-blocking, it can start P_n before A_1 has received the start signal. Of course, A_1 does not need to await A_n's start and next round signals to make sure that every partial scheduler is ready to start the next round.

Let us specify the partial schedulers formally:

$$A_1 := \mu X(a_1, c_n, d_1); d_1 \triangleleft \mathsf{start} \cdot a_1 \triangleleft \mathsf{start} \cdot a_1 \triangleright \mathsf{ack}; d_1 \triangleleft \mathsf{next} \cdot c_n \triangleright \mathsf{start}; c_n \triangleright \mathsf{next}; X\langle a_1, c_n, d_1 \rangle$$

$$A_{i+1} := \mu X(a_{i+1}, c_i, d_{i+1});\ c_i \triangleright \mathsf{start}; a_{i+1} \triangleleft \mathsf{start} \cdot d_{i+1} \triangleleft \mathsf{start} \cdot a_{i+1} \triangleright \mathsf{ack}; \qquad \forall 1 \leq i < n$$
$$c_i \triangleright \mathsf{next}; d_{i+1} \triangleleft \mathsf{next} \cdot X\langle a_{i+1}, c_i, d_{i+1} \rangle$$

The syntax '$\mu X(\tilde{x}); P$' denotes a recursive loop where P has access to the endpoints in $\tilde{x}$ and P may contain recursive calls '$X\langle \tilde{y} \rangle$' where the endpoints in $\tilde{y}$ are assigned to $\tilde{x}$ in the next round of the loop. The syntax '$x \triangleleft \ell$' denotes the output of label ℓ on x, and '$x \triangleright \ell$' denotes the input of label ℓ on x. Outputs are non-blocking, denoted '$\cdot$', whereas inputs are blocking, denoted ';'. For example, process $x \triangleleft \ell \cdot y \triangleright \ell'; P$ may receive ℓ' on y and continue as $x \triangleleft \ell \cdot P$.

Leaving the workers unspecified, we formally specify the complete scheduler as a ring of partial schedulers connected to workers:

$$Sched_n := (\boldsymbol{\nu} c_i d_i)_{1 \leq i \leq n}(\prod_{1 \leq i \leq n}(\boldsymbol{\nu} a_i b_i)(A_i \mid P_i))$$

The syntax '$(\boldsymbol{\nu} xy)P$' denotes the connection of endpoints x and y in P, and '$\prod_{i \in I} P_i$' and '$P \mid Q$' denote parallel composition. Figure 1 illustrates $Sched_6$, the scheduler for six workers.

We return to this example in Section 4, where we type check the scheduler using APCP to show that it is deadlock free.

3 APCP: Asynchronous Priority-based Classical Processes

In this section, we define APCP, a linear type system for π-calculus processes that communicate asynchronously (i.e., the output of messages is non-blocking) on connected channel endpoints. In APCP, processes may be recursive and cyclically connected. Our type system assigns to endpoints types that specify two-party protocols, in the style of binary session types [14].

APCP combines the salient features of Dardha and Gay's Priority-based Classical Processes (PCP) [8] with DeYoung *et al.*'s semantics for asynchronous communication [10], both works inspired by Curry-Howard correspondences between linear logic and session types [7, 28]. Recursion—not present in the works by Dardha and Gay and DeYoung *et al.*—is an orthogonal feature, whose syntax is inspired by the work of Toninho *et al.* [25].

As in PCP, types in APCP rely on *priority* annotations, which enable cyclic connections by ruling out circular dependencies between sessions. A key insight of our work is that asynchrony induces significant improvements in priority management: the non-blocking outputs of APCP do not need priority checks, whereas PCP's outputs are blocking and thus require priority checks.

Properties of well-typed APCP processes are *type preservation* (Theorem 2) and *deadlock freedom* (Theorem 6). This includes cyclically connected processes, which priority-annotated types guarantee free from circular dependencies that may cause deadlock.

3.1 The Process Language

We consider an asynchronous π-calculus [15, 4]. We write $x, y, z, \ldots$ to denote (channel) *endpoints* (also known as *names*), and write $\tilde{x}, \tilde{y}, \tilde{z}, \ldots$ to denote sequences of endpoints. Also, we write $i, j, k, \ldots$ to denote *labels* for choices and $I, J, K, \ldots$ to denote sets of labels. We write $X, Y, \ldots$ to denote *recursion variables*, and $P, Q, \ldots$ to denote processes.

Figure 2 (top) gives the syntax of processes. The output action '$x[y,z]$' sends a message y (an endpoint) and a continuation endpoint z along x. The input prefix '$x(y,z);P$' blocks until a message and a continuation endpoint are received on x (referred to in P as the placeholders y and z, respectively), binding y and z in P. The selection action '$x[z] \triangleleft i$' sends a label i and a continuation endpoint z along x. The branching prefix '$x(z) \triangleright \{i : P_i\}_{i \in I}$' blocks until it receives a label $i \in I$ and a continuation endpoint (reffered to in P_i as the placeholder z) on x, binding z in each P_i. Restriction '$(\boldsymbol{\nu} xy)P$' binds x and y in P, thus declaring them as the two endpoints of the same channel and enabling communication, as in Vasconcelos [27]. The process '$(P \mid Q)$' denotes the parallel composition of P and Q. The process '$\mathbf{0}$' denotes inaction. The forwarder process '$x \leftrightarrow y$' is a primitive copycat process that links together x and y. The prefix '$\mu X(\tilde{x});P$' defines a recursive loop, binding occurrences of X in P; the endpoints $\tilde{x}$ form a context for P. The recursive call '$X\langle\tilde{x}\rangle$' loops to its corresponding μX, providing the endpoints $\tilde{x}$ as

Process syntax:

$$\begin{array}{llllll} P,Q ::= x[y,z] & \text{output} & \mid\ x(y,z);P & \text{input} & & \\ \quad\mid\ x[z] \triangleleft i & \text{selection} & \mid\ x(z) \triangleright \{i : P_i\}_{i \in I} & \text{branching} & \mid\ (\boldsymbol{\nu} xy)P & \text{restriction} \\ \quad\mid\ (P \mid Q) & \text{parallel} & \mid\ \mathbf{0} & \text{inaction} & \mid\ x \leftrightarrow y & \text{forwarder} \\ \quad\mid\ \mu X(\tilde{x});P & \text{recursive loop} & \mid\ X\langle \tilde{x} \rangle & \text{recursive call} & & \end{array}$$

Structural congruence:

$$\begin{array}{rcl} P \equiv_\alpha P' \implies & P \equiv P' & x \leftrightarrow y \equiv y \leftrightarrow x \\ & P \mid Q \equiv Q \mid P & (\boldsymbol{\nu} xy)x \leftrightarrow y \equiv \mathbf{0} \\ & P \mid \mathbf{0} \equiv P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\ x,y \notin \mathrm{fn}(P) \implies & P \mid (\boldsymbol{\nu} xy)Q \equiv (\boldsymbol{\nu} xy)(P \mid Q) & (\boldsymbol{\nu} xy)\mathbf{0} \equiv \mathbf{0} \\ |\tilde{x}| = |\tilde{y}| \implies & \mu X(\tilde{x});P \equiv P\{\mu X(\tilde{y});P\{\tilde{y}/\tilde{x}\}/X\langle \tilde{y} \rangle\} & (\boldsymbol{\nu} xy)P \equiv (\boldsymbol{\nu} yx)P \\ & & (\boldsymbol{\nu} xy)(\boldsymbol{\nu} zw)P \equiv (\boldsymbol{\nu} zw)(\boldsymbol{\nu} xy)P \end{array}$$

Reduction:

$$\begin{array}{lrl} \beta_{\mathrm{ID}} & z,y \neq x \implies & (\boldsymbol{\nu} yz)(x \leftrightarrow y \mid P) \longrightarrow P\{x/z\} \\ \beta_{\otimes ⅋} & & (\boldsymbol{\nu} xy)(x[a,b] \mid y(v,z);P) \longrightarrow P\{a/v,b/z\} \\ \beta_{\oplus \&} & j \in I \implies & (\boldsymbol{\nu} xy)(x[b] \triangleleft j \mid y(z) \triangleright \{i : P_i\}_{i \in I}) \longrightarrow P_j\{b/z\} \\ \kappa_{⅋} & x \notin \tilde{v},\tilde{w} \implies & (\boldsymbol{\nu} \tilde{v}\tilde{w})(x(y,z);P \mid Q) \longrightarrow x(y,z);(\boldsymbol{\nu} \tilde{v}\tilde{w})(P \mid Q) \\ \kappa_{\&} & x \notin \tilde{v},\tilde{w} \implies & (\boldsymbol{\nu} \tilde{v}\tilde{w})(x(z) \triangleright \{i : P_i\}_{i \in I} \mid Q) \longrightarrow x(z) \triangleright \{i : (\boldsymbol{\nu} \tilde{v}\tilde{w})(P_i \mid Q)\}_{i \in I} \end{array}$$

$$\frac{(P \equiv P') \wedge (P' \longrightarrow Q') \wedge (Q' \equiv Q)}{P \longrightarrow Q} \rightarrow_{\equiv} \qquad \frac{P \longrightarrow Q}{(\boldsymbol{\nu} xy)P \longrightarrow (\boldsymbol{\nu} xy)Q} \rightarrow_{\nu} \qquad \frac{P \longrightarrow Q}{P \mid R \longrightarrow Q \mid R} \rightarrow_{\mid}$$

Figure 2: Definition of APCP's process language.

context. We only consider contractive recursion, disallowing processes with subexpressions of the form '$\mu X_1(\tilde{x});\ldots;\mu X_n(\tilde{x});X_1\langle\tilde{x}\rangle$'.

Endpoints and recursion variables are free unless otherwise stated (i.e., unless they are bound somehow). We write '$\mathrm{fn}(P)$' and '$\mathrm{frv}(P)$' for the sets of free names and free recursion variables of P, respectively. Also, we write '$P\{x/y\}$' to denote the capture-avoiding substitution of the free occurrences of y in P for x. The notation '$P\{\mu X(\tilde{y});P'/X\langle\tilde{y}\rangle\}$' denotes the substitution of occurrences of recursive calls '$X\langle\tilde{y}\rangle$' in P with the recursive loop '$\mu X(\tilde{y});P'$', which we call *unfolding* recursion. We write sequences of substitutions '$P\{x_1/y_1\}\ldots\{x_n/y_n\}$' as '$P\{x_1/y_1,\ldots,x_n/y_n\}$'.

Except for asynchrony and recursion, there are minor differences with respect to the languages of Dardha and Gay [8] and DeYoung *et al.* [10]. Unlike Dardha and Gay's, our syntax does not include empty input and output prefixes that explicitly close channels; this simplifies the type system. We also do not include the operator for replicated servers, denoted '$!x(y);P$', which is present in the works by

both Dardha and Gay and DeYoung *et al.* Although replication can be handled without difficulties, we omit it here; we prefer focusing on recursion, because it fits well with the examples we consider. We discuss further these omitted constructs in Section 3.4.

Simplifying Notation In an output '$x[y,z]$', both y and z are free; they can be bound to a continuation process using parallel composition and restriction, as in $(\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[y,z] \mid P_{a,b})$. The same applies to selection '$x[z] \triangleleft i$'. We introduce useful notations that elide the restrictions and continuation endpoints:

Notation 1 (Derivable Actions and Prefixes)**.** *We use the following syntactic sugar:*

$$\overline{x}[y] \cdot P := (\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[a,b] \mid P\{z/x\}) \qquad \overline{x} \triangleleft \ell \cdot P := (\boldsymbol{\nu} zb)(x[b] \triangleleft \ell \mid P\{z/x\})$$
$$x(y);P := x(y,z);P\{z/x\} \qquad x \triangleright \{i : P_i\}_{i \in I} := x(z) \triangleright \{i : P_i\{z/x\}\}_{i \in I}$$

Note the use of '$\cdot$' instead of '$;$' in output and selection to stress that they are non-blocking.

Operational Semantics We define a reduction relation for processes ($P \longrightarrow Q$) that formalizes how complementary actions on connected endpoints may synchronize. As usual for π-calculi, reduction relies on *structural congruence* ($P \equiv Q$), which equates the behavior of processes with minor syntactic differences; it is the smallest congruence relation satisfying the axioms in Figure 2 (middle).

Structural congruence defines the following properties of our process language. Processes are equivalent up to α-equivalence. Parallel composition is associative and commutative, with unit '$\mathbf{0}$'. The forwarder process is symmetric, and equivalent to inaction if both endpoints are bound together through restriction. A parallel process may be moved into or out of a restriction as long as the bound channels do not appear free in the moved process: this is *scope inclusion* and *scope extrusion*, respectively. Restrictions on inactive processes may be dropped, and the order of endpoints in restrictions and of consecutive restrictions does not matter. Finally, a recursive loop is equivalent to its unfolding, replacing any recursive calls with copies of the recursive loop, where the call's endpoints are pairwise substituted for the contextual endpoints of the loop (this is *equi-recursion*; see, e.g., Pierce [22]).

We can now define our reduction relation. Besides synchronizations, reduction includes *commuting conversions*, which allow pulling prefixes on free channels out of restrictions; they are not necessary for deadlock freedom, but they are usually presented in Curry-Howard interpretations of linear logic [7, 28, 8, 10]. We define the reduction relation '$P \longrightarrow Q$' by the axioms and closure rules in Figure 2 (bottom). Axioms labeled 'β' are *synchronizations* and those labeled 'κ' are commuting conversions. We write '$\longrightarrow_\beta$' for reductions derived from β-axioms, and '$\longrightarrow^*$' for the reflexive, transitive closure of '$\longrightarrow$'.

Rule β_{ID} implements the forwarder as a substitution. Rule $\beta_{\otimes ⅋}$ synchronizes an output and an input on connected endpoints and substitutes the message and continuation endpoint. Rule $\beta_{\oplus \&}$ synchronizes a selection and a branch: the received label determines the continuation process, substituting the continuation endpoint appropriately. Rule $\kappa_{⅋}$ (resp. $\kappa_{\&}$) pulls an input (resp. a branching) prefix on free channels out of enclosing restrictions. Rules $\rightarrow_{\equiv}$, $\rightarrow_{\nu}$, and $\rightarrow_{\mid}$ close reduction under structural congruence, restriction, and parallel composition, respectively.

Notice how output and selection actions send free names. This is different from the works by Dardha and Gay [8] and DeYoung *et al.* [10], where, following an internal mobility discipline [3], communication involves bound names only. As we show in the next subsection, this kind of *bound output* is derivable (cf. Theorem 1).

3.2 The Type System

APCP types processes by assigning binary session types to channel endpoints. Following Curry-Howard interpretations, we present session types as linear logic propositions (cf., e.g., Wadler [28], Caires and Pfenning [6], and Dardha and Gay [8]). We extend these propositions with recursion and *priority* annotations on connectives. Intuitively, actions typed with lower priority should be performed before those with higher priority. We write $\mathsf{o}, \kappa, \pi, \rho, \ldots$ to denote priorities, and 'ω' to denote the ultimate priority that is greater than all other priorities and cannot be increased further. That is, $\forall t \in \mathbb{N}.\ \omega > t$ and $\forall t \in \mathbb{N}.\ \omega + t = \omega$.

Duality, the cornerstone of session types and linear logic, ensures that the two endpoints of a channel have matching actions. Furthermore, dual types must have matching priority annotations. The following inductive definition of duality suffices for our tail-recursive types (cf. Gay *et al.* [11]).

Definition 1 (Session Types and Duality)**.** *The following grammar defines the syntax of* session types *A, B, followed by the dual $\overline{A}, \overline{B}$ of each type. Let $\mathsf{o} \in \mathbb{N} \cup \{\omega\}$.*

$$A, B ::= A \otimes^{\mathsf{o}} B \mid A \mathbin{⅋^{\mathsf{o}}} B \mid \oplus^{\mathsf{o}}\{i : A_i\}_{i \in I} \mid \&^{\mathsf{o}}\{i : A_i\}_{i \in I} \mid \bullet \mid \mu X.A \mid X$$
$$\overline{A}, \overline{B} ::= \overline{A} \mathbin{⅋^{\mathsf{o}}} \overline{B} \mid \overline{A} \otimes^{\mathsf{o}} \overline{B} \mid \&^{\mathsf{o}}\{i : \overline{A_i}\}_{i \in I} \mid \oplus^{\mathsf{o}}\{i : \overline{A_i}\}_{i \in I} \mid \bullet \mid \mu X.\overline{A} \mid X$$

An endpoint of type '$A \otimes^{\mathsf{o}} B$' (resp. '$A \mathbin{⅋^{\mathsf{o}}} B$') first outputs (resp. inputs) an endpoint of type A and then behaves as B. An endpoint of type '$\&^{\mathsf{o}}\{i : A_i\}_{i \in I}$' offers a choice: after receiving a label $i \in I$, the endpoint behaves as A_i. An endpoint of type '$\oplus^{\mathsf{o}}\{i : A_i\}_{i \in I}$' selects a label $i \in I$ and then behaves as A_i. An endpoint of type '$\bullet$' is closed; it does not require a priority, as closed endpoints do not exhibit behavior and thus are non-blocking. We define '$\bullet$' as a single, self-dual type for closed endpoints, following Caires [5]: the units '$\bot$' and '$\mathbf{1}$' of linear logic (used by, e.g., Caires and Pfenning [7] and Dardha and Gay [8] for session closing) are interchangeable in the absence of explicit closing.

Type '$\mu X.A$' denotes a recursive type, in which A may contain occurrences of the recursion variable 'X'. As customary, 'μ' is a binder: it induces the standard notions of α-equivalence, substitution (denoted '$A\{B/X\}$'), and free recursion variables (denoted '$\mathrm{frv}(A)$'). We work with tail-recursive, contractive types, disallowing types of the form '$\mu X_1 \ldots . \mu X_n . X_1$'. We adopt an equi-recursive view: a recursive type is equal to its unfolding. We postpone formalizing the unfolding of recursive types, as it requires additional definitions to ensure consistency of priorities upon unfolding.

The priority of a type is determined by the priority of the type's outermost connective:

Definition 2 (Priorities)**.** *For session type A, '$\mathrm{pr}(A)$' denotes its* priority*:*

$$\mathrm{pr}(A \otimes^{\mathsf{o}} B) := \mathrm{pr}(A \mathbin{⅋^{\mathsf{o}}} B) := \mathsf{o} \qquad \mathrm{pr}(\mu X.A) := \mathrm{pr}(A)$$
$$\mathrm{pr}(\oplus^{\mathsf{o}}\{i : A_i\}_{i \in I}) := \mathrm{pr}(\&^{\mathsf{o}}\{i : A_i\}_{i \in I}) := \mathsf{o} \qquad \mathrm{pr}(\bullet) := \mathrm{pr}(X) := \omega$$

The priority of '$\bullet$' and 'X' is ω: they denote "final", non-blocking actions of protocols. Although '$\otimes$' and '$\oplus$' also denote non-blocking actions, their priority is not constant: duality ensures that the priority for '$\otimes$' (resp. '$\oplus$') matches the priority of a corresponding '⅋' (resp. '&'), which denotes a blocking action.

Having defined the priority of types, we now turn to formalizing the unfolding of recursive types. Recall the intuition that actions typed with lower priority should be performed before those with higher priority. Based on this rationale, we observe that unfolding should increase the priorities of the unfolded type. This is because the actions related to the unfolded recursion should be performed *after* the prefix. The following definition *lifts* priorities in types:

Definition 3 (Lift). *For proposition A and $t \in \mathbb{N}$, we define '$\uparrow^t A$' as the* lift *operation:*

$$\begin{array}{lll} \uparrow^t (A \otimes^{\mathsf{o}} B) := (\uparrow^t A) \otimes^{\mathsf{o}+t} (\uparrow^t B) & \uparrow^t (\oplus^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \oplus^{\mathsf{o}+t}\{i : \uparrow^t A_i\}_{i\in I} & \uparrow^t \bullet := \bullet \\ \uparrow^t (A \parr^{\mathsf{o}} B) := (\uparrow^t A) \parr^{\mathsf{o}+t} (\uparrow^t B) & \uparrow^t (\&^{\mathsf{o}}\{i : A_i\}_{i\in I}) := \&^{\mathsf{o}+t}\{i : \uparrow^t A_i\}_{i\in I} & \\ \uparrow^t (\mu X.A) := \mu X.\uparrow^t (A) & \uparrow^t X := X & \end{array}$$

Henceforth, the recursive type '$\mu X.A$' and its unfolding '$A\{\uparrow^t \mu X.A/X\}$' denote the same type, where the lift $t \in \mathbb{N}$ of the unfolded recursive calls depends on the context in which the type appears.

Typing Rules The typing rules of APCP ensure that actions with lower priority are performed before those with higher priority (cf. Dardha and Gay [8]). To this end, they enforce the following laws:

1. an action with priority o must be prefixed only by inputs and branches with priority strictly smaller than o—this law does not hold for output and selection, as they are not prefixes;
2. dual actions leading to synchronizations must have equal priorities (cf. Def. 1).

Judgments are of the form '$P \vdash \Omega; \Gamma$', where P is a process, Γ is a context that assigns types to channels ('$x{:}A$'), and Ω is a context that assigns natural numbers to recursion variables ('$X{:}n$'). The intuition behind the latter context is that it ensures the amount of context endpoints to concur between recursive definitions and calls. Both contexts Γ and Ω obey *exchange*: assignments may be silently reordered. Γ is *linear*, disallowing *weakening* (i.e., all assignments must be used) and *contraction* (i.e., assignments may not be duplicated). Ω allows weakening and contraction, because a recursive definition does not necessarily require a recursive call although it may be called more than once. The empty context is written '$\emptyset$'. We write '$\mathrm{pr}(\Gamma)$' to denote the least priority of all types in Γ. Notation '$(x_i{:}A_i)_{i\in I}$' denotes indexing of assignments by I. We write '$\uparrow^t \Gamma$' to denote the component-wise extension of lift to typing contexts.

Figure 3 (top) gives the typing rules. Typing is closed under structural congruence; we sometimes use this explicitly in typing derivations in the form of a rule '$\equiv$'. Axiom 'EMPTY' types an inactive process with no endpoints. Rule '$\bullet$' silently adds a closed endpoint to the typing context. Axiom 'ID' types forwarding between endpoints of dual type. Rule 'MIX' types the parallel composition of two processes that do not share assignments on the same endpoints. Rule 'CYCLE' removes two endpoints of dual type from the context by adding a restriction on them. Note that a single application of 'MIX' followed by 'CYCLE' coincides with the usual rule 'CUT' in type systems based on linear logic [7, 28]. Axiom '$\otimes$' types an output action; this rule does not have premises to provide a continuation process, leaving the free endpoints to be bound to a continuation process using 'MIX' and 'CYCLE'. Similarly, axiom '$\oplus$' types an unbounded selection action. Priority checks are confined to rules '$\parr$' and '&', which type an input and a branching prefix, respectively. In both cases, the used endpoint's priority must be lower than the priorities of the other types in the continuation's typing context.

Rule 'REC' types a recursive definition by eliminating a recursion variable from the recursion context whose value concurs with the size of the typing context, where contractiveness is guaranteed by requiring that the eliminated recursion variable may not appear unguarded in each of the context's types. Axiom 'VAR' types a recursive call by adding a recursion variable to the context with the amount of introduced endpoints. As mentioned before, the value of the introduced and consequently eliminated recursion variable is crucial in ensuring that a recursion is called with the same amount of channels as required by its definition.

Let us compare our typing system to that of Dardha and Gay [8] and DeYoung *et al.* [10]. Besides our support for recursion, the main difference is that our rules for output and selection are axioms.

$$\frac{}{\mathbf{0} \vdash \Omega; \emptyset}\ \textsc{Empty} \qquad \frac{P \vdash \Omega; \Gamma}{P \vdash \Omega; \Gamma, x{:}\bullet}\ \bullet \qquad \frac{}{x \leftrightarrow y \vdash \Omega; x{:}\overline{A}, y{:}A}\ \textsc{Id}$$

$$\frac{P \vdash \Omega; \Gamma \quad Q \vdash \Omega; \Delta}{P \mid Q \vdash \Omega; \Gamma, \Delta}\ \textsc{Mix} \qquad \frac{P \vdash \Omega; \Gamma, x{:}A, y{:}\overline{A}}{(\boldsymbol{\nu} xy)P \vdash \Omega; \Gamma}\ \textsc{Cycle}$$

$$\frac{}{x[y,z] \vdash \Omega; x{:}A \otimes^{\mathsf{o}} B, y{:}\overline{A}, z{:}\overline{B}}\ \otimes \qquad \frac{P \vdash \Omega; \Gamma, y{:}A, z{:}B \quad \mathsf{o} < \mathrm{pr}(\Gamma)}{x(y,z); P \vdash \Omega; \Gamma, x{:}A \parr^{\mathsf{o}} B}\ \parr$$

$$\frac{j \in I}{x[z] \triangleleft j \vdash \Omega; x{:}\oplus^{\mathsf{o}}\{i : A_i\}_{i \in I}, z{:}\overline{A_j}}\ \oplus \qquad \frac{\forall i \in I.\ P_i \vdash \Omega; \Gamma, z{:}A_i \quad \mathsf{o} < \mathrm{pr}(\Gamma)}{x(z) \triangleright \{i : P_i\}_{i \in I} \vdash \Omega; \Gamma, x{:}\&^{\mathsf{o}}\{i : A_i\}_{i \in I}}\ \&$$

$$\frac{P \vdash \Omega, X{:}|I|; (x_i{:}A_i)_{i \in I} \quad \forall i \in I.\ A_i \neq X}{\mu X((x_i)_{i \in I}); P \vdash \Omega; (x_i{:}\mu X.A_i)_{i \in I}}\ \textsc{Rec} \qquad \frac{}{X\langle (x_i)_{i \in I} \rangle \vdash \Omega, X{:}|I|; (x_i{:}X)_{i \in I}}\ \textsc{Var}$$

$$\frac{P \vdash \Omega; \Gamma, y{:}A, x{:}B}{\overline{x}[y] \cdot P \vdash \Omega; \Gamma, x{:}A \otimes^{\mathsf{o}} B}\ \otimes^\star \qquad \frac{P \vdash \Omega; \Gamma, x{:}A_j \quad j \in I}{\overline{x} \triangleleft j \cdot P \vdash \Omega; \Gamma, x{:}\oplus^{\mathsf{o}}\{i : A_i\}_{i \in I}}\ \oplus^\star \qquad \frac{P \vdash \Omega; \Gamma \quad t \in \mathbb{N}}{P \vdash \Omega; \uparrow^t \Gamma}\ \textsc{Lift}$$

Figure 3: The typing rules of APCP (top) and admissible rules (bottom).

This makes priority checking much simpler for APCP than for Dardha and Gay's PCP: our outputs and selections have no typing context to check priorities against, and types for closed endpoints have no priority at all. Although DeYoung *et al.*'s output and selection actions are atomic too, their corresponding rules are similar to the rules of Dardha and Gay: the rules require continuation processes as premises, immediately binding the sent endpoints.

As anticipated, the binding of output and selection actions to continuation processes (Notation 1) is derivable in APCP. The corresponding typing rules in Figure 3 (bottom) are admissible using 'MIX' and 'CYCLE'. Note that it is not necessary to include rules for the sugared input and branching in Notation 1, because they rely on name substitution only and typing is closed under structural congruence and thus name substitution. Figure 3 (bottom) also includes an admissible rule 'LIFT' that lifts a process' priorities.

Theorem 1. *The rules '$\otimes^\star$', '$\oplus^\star$', and 'LIFT' in Figure 3 (bottom) are admissible.*

Proof. We show the admissibility of rules $\otimes^\star$ and $\oplus^\star$ by giving their derivations in Figure 4 (omitting the recursion context). The rule 'LIFT' is admissible, because $P \vdash \Omega; \Gamma$ implies $P \vdash \Omega; \uparrow^t \Gamma$ (cf. Dardha and Gay [8]), by simply increasing all priorities in the derivation of P by t. □

Theorem 1 highlights how APCP's asynchrony uncovers a more primitive, lower-level view of message-passing. In the next subsection we discuss deadlock freedom, which follows from a correspondence between reduction and the removal of 'CYCLE' rules from typing derivations. In the case of APCP, this requires care: binding output and selection actions to continuation processes leads to applications of 'CYCLE' not immediately corresponding to reductions.

$$\dfrac{P \vdash \Gamma, y{:}A, x{:}B}{\overline{x}[y] \cdot P \vdash \Gamma, x{:}A \otimes^\circ B}\otimes^\star \quad \Rightarrow \quad \dfrac{\dfrac{\dfrac{}{x[a,b] \vdash x{:}A \otimes^\circ B, a{:}\overline{A}, b{:}\overline{B}}\otimes \quad \dfrac{P \vdash \Gamma, y{:}A, x{:}B}{P\{z/x\} \vdash \Gamma, y{:}A, z{:}B}\equiv}{x[a,b] \mid P\{z/x\} \vdash \Gamma, x{:}A \otimes^\circ B, y{:}A, a{:}\overline{A}, z{:}B, b{:}\overline{B}}\textsc{Mix}}{\underbrace{(\boldsymbol{\nu} ya)(\boldsymbol{\nu} zb)(x[a,b] \mid P\{z/x\})}_{\overline{x}[y]\cdot P \text{ (cf. Notation 1)}} \vdash \Gamma, x{:}A \otimes^\circ B}\textsc{Cycle}^2$$

$$\dfrac{P \vdash \Gamma, x{:}A_j \quad j \in I}{\overline{x} \triangleleft j \cdot P \vdash \Gamma, x{:}\oplus^\circ\{i : A_i\}_{i \in I}}\oplus^\star \quad \Rightarrow \quad \dfrac{\dfrac{\dfrac{j \in I}{x[b] \triangleleft j \vdash x{:}\oplus^\circ\{i : A_i\}_{i \in I}, b{:}\overline{A_j}}\oplus \quad \dfrac{P \vdash \Gamma, x{:}A_j}{P\{z/x\} \vdash \Gamma, z{:}A_j}\equiv}{x[b] \triangleleft j \mid P\{z/x\} \vdash \Gamma, x{:}\oplus^\circ\{i : A_i\}_{i \in I}, z{:}A_j, b{:}\overline{A_j}}\textsc{Mix}}{\underbrace{(\boldsymbol{\nu} zb)(x[b] \triangleleft j \mid P\{z/x\})}_{\overline{x}\triangleleft j\cdot P \text{ (cf. Notation 1)}} \vdash \Gamma, x{:}\oplus^\circ\{i : A_i\}_{i \in I}}\textsc{Cycle}$$

Figure 4: Proof that rules '$\otimes^\star$' and '$\oplus^\star$' are admissible (cf. Theorem 1).

3.3 Type Preservation and Deadlock Freedom

Well-typed processes satisfy protocol fidelity, communication safety, and deadlock freedom. All these properties follow from *type preservation* (also known as *subject reduction*), which ensures that reduction preserves typing. In contrast to Caires and Pfenning [7] and Wadler [28], where type preservation corresponds to the elimination of (top-level) applications of rule CUT, in APCP it corresponds to the elimination of (top-level) applications of rule CYCLE.

Theorem 2 (Type Preservation)**.** *If* $P \vdash \Omega; \Gamma$ *and* $P \longrightarrow Q$*, then* $Q \vdash \Omega; \uparrow^t \Gamma$ *for* $t \in \mathbb{N}$*.*

Proof. By induction on the reduction $\longrightarrow$, analyzing the last applied rule (Fig. 2 (bottom)). The cases of the closure rules $\rightarrow_\equiv$, $\rightarrow_\nu$, and $\rightarrow_|$ easily follow from the IH. The key cases are the β- and κ-rules. Figure 5 shows two representative instances (eluding the recursion context Ω): rule $\beta_{\otimes \parr}$ (top), a synchronization, and rule $\kappa_\parr$ (bottom), a commuting conversion. Note how, in the case of rule $\kappa_\parr$, the lift $\uparrow^t$ ensures consistent priority checks. □

Protocol fidelity ensures that processes respect their intended (session) protocols. Communication safety ensures the absence of communication errors and mismatches in processes. Correct typability gives a static guarantee that a process conforms to its ascribed session protocols; type preservation gives a dynamic guarantee. Because session types describe the intended protocols and error-free exchanges, type preservation entails both protocol fidelity and communication safety. We refer the curious reader to the early work by Honda *et al.* [16] for a detailed account, which shows by contradiction that well-typed processes do not reduce to so-called error processes. This is a well-known and well-understood result.

In what follows, we consider a process to be deadlocked if it is not the inactive process and cannot reduce. Our deadlock freedom result for APCP adapts that for PCP [8], which involves three steps:

1. First, CYCLE-elimination states that we can remove all applications of CYCLE in a typing derivation without affecting the derivation's assumptions and conclusion.
2. Only the removal of *top-level* CYCLEs captures the intended process semantics, as the removal of other CYCLEs corresponds to reductions behind prefixes which is not allowed [28, 8]. Therefore, the second step is *top-level deadlock freedom*, which states that a process with a top-level CYCLE reduces until there are no top-level CYCLEs left.

$$\dfrac{\dfrac{}{x[a,b] \vdash x{:}A \otimes^{\circ} B, a{:}\overline{A}, b{:}\overline{B}}\otimes \quad \dfrac{P \vdash \Gamma, v{:}\overline{A}, z{:}\overline{B}}{y(v,z).P \vdash \Gamma, y{:}\overline{A} \parr^{\circ} \overline{B}}\parr}{(\boldsymbol{\nu}xy)(x[a,b] \mid y(v,z).P) \vdash \Gamma, a{:}\overline{A}, b{:}\overline{B}}\ \textsc{Mix}+\textsc{Cycle} \quad \longrightarrow \quad \dfrac{P \vdash \Gamma, v{:}\overline{A}, z{:}\overline{B}}{P\{a/v,b/z\} \vdash \Gamma, a{:}\overline{A}, b{:}\overline{B}}\equiv$$

Below, the contexts Γ' and Δ' together contain $\tilde{v}$ and $\tilde{w}$, i.e. $\Gamma', \Delta' = (v_i{:}C_i)_{v_i \in \tilde{v}}, (w_i{:}\overline{C_i})_{w_i \in \tilde{w}}$.

$$\dfrac{\dfrac{P \vdash \Gamma, \Gamma', y{:}A, z{:}B \quad \mathsf{o} < \mathrm{pr}(\Gamma)}{x(y,z).P \vdash \Gamma, \Gamma', x{:}A \parr^{\circ} B}\parr \quad Q \vdash \Delta, \Delta'}{(\boldsymbol{\nu}\tilde{v}\tilde{w})(x(y,z).P \mid Q) \vdash \Gamma, \Delta, x{:}A \parr^{\circ} B}\ \textsc{Mix}+\textsc{Cycle}^{*}$$

$$\longrightarrow$$

$$\dfrac{\dfrac{\dfrac{P \vdash \Gamma, \Gamma', y{:}A, z{:}B \quad Q \vdash \Delta, \Delta'}{(\boldsymbol{\nu}\tilde{v}\tilde{w})(P \mid Q) \vdash \Gamma, \Delta, y{:}A, z{:}B}\ \textsc{Mix}+\textsc{Cycle}^{*}}{(\boldsymbol{\nu}\tilde{v}\tilde{w})(P \mid Q) \vdash {\uparrow^{\mathsf{o}+1}}\Gamma, {\uparrow^{\mathsf{o}+1}}\Delta, y{:}{\uparrow^{\mathsf{o}+1}}A, z{:}{\uparrow^{\mathsf{o}+1}}B}\ \textsc{Lift} \quad \mathsf{o} < \mathrm{pr}({\uparrow^{\mathsf{o}+1}}\Gamma, {\uparrow^{\mathsf{o}+1}}\Delta)}{x(y,z).(\boldsymbol{\nu}\tilde{v}\tilde{w})(P \mid Q) \vdash {\uparrow^{\mathsf{o}+1}}\Gamma, {\uparrow^{\mathsf{o}+1}}\Delta, x{:}({\uparrow^{\mathsf{o}+1}}A) \parr^{\mathsf{o}} ({\uparrow^{\mathsf{o}+1}}B)}\parr$$

Figure 5: Type Preservation (cf. Theorem 2) in rules $\beta_{\otimes\parr}$ (top) and $\kappa_{\parr}$ (bottom).

3. Third, deadlock freedom follows for processes typable under empty contexts.

Here, we address cycle-elimination and top-level deadlock-freedom in one proof.

As mentioned before, binding APCP's asynchronous outputs and selections to continuations involves additional, low-level uses of CYCLE, which we cannot eliminate through process reduction. Therefore, we establish top-level deadlock freedom for *live processes* (Theorem 4). A process is live if it is equivalent to a restriction on *active names* that perform unguarded actions. This way, e.g., in '$x[y,z]$' the name x is active, but y and z are not.

Definition 4 (Active Names)**.** *The* set of active names *of P, denoted* '$\mathrm{an}(P)$'*, contains the (free) names that are used for unguarded actions (output, input, selection, branching):*

$$\begin{array}{lll}
\mathrm{an}(x[y,z]) := \{x\} & \mathrm{an}(x(y,z).P) := \{x\} & \mathrm{an}(\mathbf{0}) := \emptyset \\
\mathrm{an}(x[z] \triangleleft j) := \{x\} & \mathrm{an}(x(z) \triangleright \{i : P_i\}_{i \in I}) := \{x\} & \mathrm{an}(x \leftrightarrow y) := \{x,y\} \\
\mathrm{an}(P \mid Q) := \mathrm{an}(P) \cup \mathrm{an}(Q) & \mathrm{an}(\mu X(\tilde{x}); P) := \mathrm{an}(P) & \\
\mathrm{an}((\boldsymbol{\nu}xy)P) := \mathrm{an}(P) \setminus \{x,y\} & \mathrm{an}(X\langle \tilde{x} \rangle) := \emptyset &
\end{array}$$

Definition 5 (Live Process)**.** *A process P is* live*, denoted* '$\mathrm{live}(P)$'*, if there are names x,y and process P' such that $P \equiv (\boldsymbol{\nu}xy)P'$ with $x,y \in \mathrm{an}(P')$.*

We additionally need to account for recursion: as recursive definitions do not entail reductions, we must fully unfold them before eliminating CYCLEs.

Lemma 3 (Unfolding)**.** *If $P \vdash \Omega; \Gamma$, then there is process $P^{\star}$ such that $P^{\star} \equiv P$ and $P^{\star}$ is not of the form '$\mu X(\tilde{x}); Q$' and $P^{\star} \vdash \Omega; \Gamma$.*

Proof. By induction on the amount n of consecutive recursive definitions prefixing P, such that P is of the form '$\mu X_1(\tilde{x}); \ldots; \mu X_n(\tilde{x}); Q$'. If $n = 0$, the thesis follows immediately by letting $P^{\star} := P$.

$$\dfrac{}{X\langle (y_i)_{i\in I}\rangle \vdash \Omega', X{:}|I|; (y_i{:}X)_{i\in I}}\ \textsc{Var}$$
$$\vdots \qquad \iddots$$
$$\dfrac{\dfrac{\qquad\qquad}{Q \vdash \Omega, X{:}|I|; (x_i{:}A_i)_{i\in I}}\ \cdots}{\mu X((x_i)_{i\in I}); Q \vdash \Omega; (x_i{:}\mu X.A_i)_{i\in I}}\ \textsc{Rec}$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$\dfrac{}{X\langle (y_i)_{i\in I}\rangle \vdash \Omega'', X{:}|I|; (y_i{:}X)_{i\in I}}\ \textsc{Var}$$
$$\vdots \qquad \iddots$$
$$\dfrac{\dfrac{\dfrac{\dfrac{\qquad\qquad}{Q \vdash \Omega', X{:}|I|; (x_i{:}A_i)_{i\in I}}\ \cdots}{Q\{(y_i)_{i\in I}/(x_i)_{i\in I}\} \vdash \Omega', X{:}|I|; (y_i{:}A_i)_{i\in I}}\ \equiv}{\mu X((y_i)_{i\in I}); Q\{(y_i)_{i\in I}/(x_i)_{i\in I}\} \vdash \Omega'; (y_i{:}\mu X.A_i)_{i\in I}}\ \textsc{Rec} \qquad t \geq \max_{\mathsf{pr}}(A_i)_{i\in I}}{\mu X((y_i)_{i\in I}); Q\{(y_i)_{i\in I}/(x_i)_{i\in I}\} \vdash \Omega'; (y_i{:}\uparrow^t \mu X.A_i)_{i\in I}}\ \textsc{Lift}$$
$$\vdots \qquad \iddots$$
$$\dfrac{\qquad\qquad}{R \vdash \Omega; (x_i{:}A_i\{\uparrow^t \mu X.A_i/X\})_{i\in I}}\ \cdots$$

Figure 6: Typing recursion before (top) and after (bottom) unfolding (cf. Lemma 3).

Otherwise, $n \geq 1$. Then there are X, Q such that $P = \mu X((x_i)_{i\in I}); Q$. By inversion of typing rule REC, $P \vdash \Omega; (x_i{:}\mu X.A_i)_{i\in I}$. Generally speaking, such typing derivations have the shape as in Figure 6 (top), with zero or more VAR-axioms on X appearing at the top. We use structural congruence (Fig. 2 (middle)) to unfold the recursion in P, obtaining the process $R := Q\{\mu X((y_i)_{i\in I}); Q\{(y_i)_{i\in I}/(x_i)_{i\in I}\}/X\langle (y_i)_{i\in I}\rangle\} \equiv P$.

We can type R by taking the derivation of P (cf. Figure 6 (top)), removing the final application of the REC-rule and replacing any uses of the VAR-axiom on X by a copy of the original derivation, applying α-conversion where necessary. Moreover, we lift the priorities of all types by at least the highest priority occurring in any type in Γ using the LIFT-rule, ensuring that priority conditions on typing rules remain valid; we explicitly use *at least* the highest priority, as the context of connected endpoints may lift the priorities in dual types even more. Writing the highest priority in Γ as '$\max_{\mathsf{pr}}(\Gamma)$', the resulting proof is of the shape in Figure 6 (bottom). Since types are equi-recursive, $A_i\{\uparrow^t \mu X.A_i/X\} = A_i$ for every $i \in I$. Hence, $(y_i{:}A_i\{\uparrow^t \mu X.A_i/X\})_{i\in I} = \Gamma$. Thus, the above is a valid derivation of $R \vdash \Omega; \Gamma$.

The rules applied after LIFT in the derivation of R in Figure 6 (bottom) are the same as those applied after VAR and before REC in the derivation of P in Figure 6 (top) before unfolding. By the assumption that recursion is contractive, there must be an application of a rule other than REC in this part of the derivation. Therefore, the application of REC in the derivation of R is not part of a possible sequence of RECs in the last-applied rules of this derivation. Hence, since we removed the final application of REC in the derivation of P, the size of this sequence of RECs is $n-1$, i.e. R is prefixed by $n-1$ recursive definitions. Thus, we apply the IH to find a process $P^\star$ not prefixed by recursive definitions s.t. $P^\star \equiv R \equiv P \vdash \Omega; \Gamma$. □

Dardha and Gay's top-level deadlock freedom result concerns a sequence of reduction steps that reaches a process that is not live anymore [8]. In our case, top-level deadlock freedom concerns a single reduction step only, because recursive processes might stay live across reductions forever.

Theorem 4 (Top-Level Deadlock Freedom)**.** *If $P \vdash \emptyset; \Gamma$ and $\mathrm{live}(P)$, then there is process Q such that $P \longrightarrow Q$.*

Proof. By structural congruence (Fig. 2 (middle)), there is $P_c = (\boldsymbol{\nu} x_i y_i)_{i \in I}(\boldsymbol{\nu} \tilde{n}\tilde{m})P_m$ such that $P_c \equiv P$, with $P_m = \prod_{k \in K} P_k$ and $P_m \vdash \emptyset; \Lambda, (x_i{:}A_i, y_i{:}\overline{A_i})_{i \in I}$ s.t. for every $i \in I$, x_i and y_i are active names in P_m, and Λ consists of Γ and the channels $\tilde{n}, \tilde{m}$ which are dually typed pairs of endpoints of which at least one is inactive in P_m. Because P is live, there is always at least one pair x_i, y_i.

Next, we take the $j \in I$ s.t. A_j has the least priority, i.e. $\forall i \in I \setminus \{j\}.\ \mathrm{pr}(A_j) \leq \mathrm{pr}(A_i)$. If there are multiple to choose from, any suffices. The rest of the analysis depends on whether there is an endpoint z of input/branching type in Γ with lower priority than $\mathrm{pr}(A_j)$. We thus distinguish the two cases below. Note that output/selection types in Γ are associated with non-blocking actions and can be safely ignored.

- If there is such z, assume w.l.o.g. it is of input type. The input on z cannot be prefixed by an input/branch on another endpoint, because then that other endpoint would have a type with lower priority than z. Hence, there is $k' \in K$ s.t. $P_{k'} = z(u,v); P'_{k'}$. We thus apply communicating conversion $\kappa_{⅋}$ to find Q such that $P \longrightarrow Q$:

$$P \equiv (\boldsymbol{\nu} x_i y_i)_{i \in I}(\boldsymbol{\nu} \tilde{n}\tilde{m})(\textstyle\prod_{k \in K \setminus \{k'\}} P_k \mid z(u,v); P'_{k'}) \longrightarrow z(u,v); (\boldsymbol{\nu} x_i y_i)_{i \in I}(\boldsymbol{\nu} \tilde{n}\tilde{m})(\textstyle\prod_{k \in K \setminus \{k'\}} P_k \mid P'_{k'}) = Q$$

- If there is no such z, we continue with $x_j{:}A_j$ and $y_j{:}\overline{A_j}$. In case there is $k' \in K$ s.t. $P_k \equiv u \leftrightarrow v$ with $u \in \{x_j, y_j\}$, the reduction is trivial by $\rightarrow_{\mathrm{ID}}$; we w.l.o.g. assume there is no such k'.

  By duality, A_j and $\overline{A_j}$ have the same priority, so priority checks in typing derivations prevent an input/branching prefix on x_j (resp. y_j) from blocking an output/selection on y_j (resp. x_j). Hence, x_j and y_j appear in separate parallel components of P_m, i.e. $P_m = P_{x_j} \mid P_{y_j} \mid P_R$ s.t.

$$P_{x_j} \vdash \emptyset; \Lambda_{x_j}, x_j{:}A_j\ , \quad P_{y_j} \vdash \emptyset; \Lambda_{y_j}, y_j{:}\overline{A_j}\ , \text{ and } \quad P_R \vdash \emptyset; \Lambda_R\ ,$$

  where $\Lambda_{x_j}, \Lambda_{y_j}, \Lambda_R, x_j{:}A_j, y_j{:}\overline{A_j} = \Lambda, (x_i{:}A_i, y_i{:}\overline{A_i})_{i \in I}$.

  By Lemma 3 (unfolding), $P_{x_j} \equiv P^\star_{x_j}$ and $P_{y_j} \equiv P^\star_{y_j}$ s.t. $P^\star_{x_j}$ and $P^\star_{y_j}$ are not prefixed by recursive definitions and $P^\star_{x_j} \vdash \emptyset; \Lambda_{x_j}, x_j{:}A_j$ and $P^\star_{y_j} \vdash \emptyset; \Lambda_{y_j}, y_j{:}\overline{A_j}$. We take the unfolded form of A_j: by the contractiveness of recursive types, A_j has at least one connective. We w.l.o.g. assume that A_j is an input or branching type, i.e. either (a) $A_j = B ⅋^{\mathsf{o}} C$ or (b) $A_j = \&^{\mathsf{o}}\{l : B_l\}_{l \in L}$.

  Since $\mathrm{pr}(A_j) = \mathsf{o}$ is the least of the priorities in Γ, we know that either (in case a) $P^\star_{x_j} \equiv x_j(v,z).Q_{x_j}$ or (in case b) $P^\star_{x_j} \equiv x_j(z) \triangleright \{l : Q^l_{x_j}\}_{l \in L}$. Moreover, since either (in case a) $\overline{A_j} = \overline{B} \otimes^{\mathsf{o}} \overline{C}$ or (in case b) $\overline{A_j} = \oplus^{\mathsf{o}}\{l : \overline{B_l}\}_{l \in L}$, we have that either (in case a) $P^\star_{y_j} \equiv y_j[a,b] \mid Q_{y_j}$ or (in case b) $P^\star_{y_j} \equiv y_j[b] \triangleleft l^\star \mid Q_{y_j}$ for $l^\star \in L$. In case (a), let $Q'_{x_j} := Q_{x_j}\{a/v, b/z\}$; in case (b), let $Q'_{x_j} := Q^{l^\star}_{x_j}\{b/z\}$. Then, (in case a) by reduction $\beta_{\otimes ⅋}$ or (in case b) by reduction $\beta_{\oplus \&}$,

$$\begin{aligned} P \equiv (\boldsymbol{\nu} x_i y_i)_i(\boldsymbol{\nu} \tilde{n}\tilde{m})(P^\star_{x_j} \mid P^\star_{y_j} \mid P_R) &\equiv (\boldsymbol{\nu} x_i y_i)_{i \setminus j}(\boldsymbol{\nu} \tilde{n}\tilde{m})((\boldsymbol{\nu} x_j y_j)(P^\star_{x_j} \mid P^\star_{y_j}) \mid P_R) \\ &\longrightarrow (\boldsymbol{\nu} x_i y_i)_{i \setminus j}(\boldsymbol{\nu} \tilde{n}\tilde{m})(Q'_{x_j} \mid Q_{y_j} \mid P_R). \end{aligned}$$ □

Our deadlock freedom result concerns processes typable under empty contexts (as in, e.g., Caires and Pfenning [7] and Dardha and Gay [8]). This way, the reduction guaranteed by Theorem 4 corresponds to a synchronization (β-rule), rather than a commuting conversion (κ-rule). We first need a lemma which ensures that non-live processes typable under empty contexts do not contain actions or prefixes.

Lemma 5. *If $P \vdash \emptyset; \emptyset$ and P is not live, then P contains no actions or prefixes whatsoever.*

Proof. Suppose, for contradiction, that P does contain actions or prefixes. For example, P contains some subterm $x(y,z);P'$. Because $P \vdash \emptyset;\emptyset$, there must be a restriction on x in P binding it with, e.g., x'. Now, x' does not appear in P', because the type of x in the derivation of $P \vdash \emptyset;\emptyset$ must be lower than the types of the endpoints in P', and by duality the types of x and x' have equal priority. Hence, there is some Q s.t. $P \equiv (\boldsymbol{\nu}\tilde{u}\tilde{v})(\boldsymbol{\nu}xx')(x(y,z);P' \mid Q)$ where $x' \in \mathrm{fn}(Q)$. There are two cases for the appearance of x' in Q: (1) not prefixed, or (2) prefixed.

- In case (1), $x' \in \mathrm{an}(Q)$, so the restriction on x,x' in P is on a pair of active names, contradicting the fact that P is not live.
- In case (2), x' appears in Q behind at least one prefix. For example, Q contains some subterm $a(b,c);Q'$ where $x' \in \mathrm{fn}(Q')$. Again, a must be bound in P to, e.g., a'. Through similar reasoning as above, we know that a' does not appear in Q'. Moreover, the type of a must have lower priority than the type of x', so by duality the type of a' must have lower priority than the type of x. So, a' also does not appear in P'. Hence, there is R s.t. $P \equiv (\boldsymbol{\nu}\tilde{u}\tilde{v})(\boldsymbol{\nu}aa')(\boldsymbol{\nu}xx')(x(y,z);P' \mid a(b,c);Q' \mid R)$ where $a' \in \mathrm{fn}(R)$.

  Now, the case split on whether a' appears prefixed in R or not repeats, possibly finding new names that prefix the current name again and again following case (2). However, process terms are finite in size, so we know that at some point there cannot be an additional parallel component in P to bind the new name, contradicting the existence of the newly found prefix. Hence, eventually case (1) will be reached, uncovering a restriction on a pair of active names and contradicting the fact that P is not live.

In conclusion, the assumption that there are actions or prefixes in P leads to a contradiction. Hence, P contains no actions or prefixes whatsoever. □

We now state our deadlock freedom result:

Theorem 6 (Deadlock Freedom)**.** *If $P \vdash \emptyset;\emptyset$, then either $P \equiv \mathbf{0}$ or $P \longrightarrow_\beta Q$ for some Q.*

Proof. The analysis depends on whether P is live or not.

- If P is not live, then, by Lemma 5, it does not contain any actions or prefixes. Any recursive loops in P are thus of the form '$\mu X_1();\ldots;\mu X_n();\mathbf{0}$': contractiveness requires recursive calls to be prefixed by inputs/branches or bound to parallel outputs/selections, of which there are none. Hence, we can use structural congruence to rewrite each recursive loop in P to $\mathbf{0}$ by unfolding, yielding $P' \equiv P$. The remaining derivation of P' only contains applications of Empty, Mix, $\bullet$, or Cycle on closed endpoints. It follows easily that $P \equiv P' \equiv \mathbf{0}$.
- If P is live, by Theorem 4 there is Q s.t. $P \longrightarrow Q$. Moreover, P does not have free names, for otherwise it would not be typable under empty context. Because commuting conversions apply only to free names, this means $P \longrightarrow_\beta Q$. □

3.4 Explicit Closing and Replicated Servers

As already mentioned, our presentation of APCP does not include explicit closing and replicated servers. We briefly discuss what APCP would look like if we were to include these constructs.

We achieve explicit closing by adding empty outputs '$x[]$' and empty inputs '$x();P$' to the syntax of Figure 2 (top). We also add the synchronization '$\beta_{\mathbf{1}\bot}$' and the commuting conversion '$\kappa_\bot$' in Figure 7 (bottom). At the level of types, we replace the conflated type '$\bullet$' with '$\mathbf{1}^{\circ}$' and '$\bot^{\circ}$', associated to empty

$$\frac{}{x[] \vdash \Omega; x{:}\mathbf{1}^{\circ}}\ \mathbf{1} \qquad \frac{P \vdash \Omega; \Gamma \quad \mathsf{o} < \mathrm{pr}(\Gamma)}{x(); P \vdash \Omega; \Gamma, x{:}\bot^{\circ}}\ \bot$$

$$\frac{}{?x[y] \vdash \Omega; x{:}?^{\circ}A, y{:}\overline{A}}\ ? \qquad \frac{P \vdash \Omega; ?\Gamma, y{:}A \quad \mathsf{o} < \mathrm{pr}(?\Gamma)}{!x(y); P \vdash \Omega; ?\Gamma, x{:}!^{\circ}A}\ !$$

$$\frac{P \vdash \Omega; \Gamma}{P \vdash \Omega; \Gamma, x{:}?^{\circ}A}\ \mathrm{W} \qquad \frac{P \vdash \Omega; \Gamma, x{:}?^{\circ}A, x'{:}?^{\kappa}A \quad \pi = \min(\mathsf{o}, \kappa)}{P\{x/x'\} \vdash \Omega; \Gamma, x{:}?^{\pi}A}\ \mathrm{C} \qquad \frac{P \vdash \Omega; \Gamma, y{:}A}{?\overline{x}[y] \cdot P \vdash \Omega; \Gamma, x{:}?^{\circ}A}\ ?^{\star}$$

$$\begin{array}{lll}
\beta_{\mathbf{1}\bot} & & (\boldsymbol{\nu}xy)(x[] \mid y(); P) \longrightarrow P \\
\beta_{?!} & & (\boldsymbol{\nu}xy)(?x[a] \mid !y(v); P \mid Q) \longrightarrow P\{a/v\} \mid (\boldsymbol{\nu}xy)(!y(v); P \mid Q) \\
\kappa_{\bot} & x \notin \tilde{v}, \tilde{w} \implies & (\boldsymbol{\nu}\tilde{v}\tilde{w})(x(); P \mid Q) \longrightarrow x(); (\boldsymbol{\nu}\tilde{v}\tilde{w})(P \mid Q) \\
\kappa_{!} & x \notin \tilde{v}, \tilde{w} \implies & (\boldsymbol{\nu}\tilde{v}\tilde{w})(!x(y); P \mid Q) \longrightarrow !x(y); (\boldsymbol{\nu}\tilde{v}\tilde{w})(P \mid Q)
\end{array}$$

Figure 7: Typing rules for explicit closing and replicated servers.

outputs and empty inputs, respectively. Note that we do need priority annotations on types for closed endpoints now, because the empty input is blocking and thus requires priority checks. In the type system of Figure 3 (top), we replace rule '$\bullet$' with the rules '$\mathbf{1}$' and '$\bot$' in Figure 7 (top).

For replicated servers, we add client requests '$?x[y]$' and servers '$!x(y); P$', typed '$?^{\circ}A$' and '$!^{\circ}A$', respectively. We include syntactic sugar for binding client requests to continuations as in Notation 1: '$?\overline{x}[y] \cdot P := (\boldsymbol{\nu}ya)(?x[a] \mid P)$'. New reduction rules are in Figure 7 (bottom): synchronization rule '$\beta_{?!}$', connecting a client and a server and spawns a copy of the server, and commuting conversion '$\kappa_{!}$'. Also, we add a structural congruence axiom to clean up unused servers: $(\boldsymbol{\nu}xz)(!x(y); P) \equiv \mathbf{0}$. In the type system, we add rules '?', '!', 'W' and 'C' in Figure 7 (top); the former two are for typing client requests and servers, respectively, and the latter two are for connecting to a server without requests and for multiple requests, respectively. In rule '!', notation '$?\Gamma$' means that every type in Γ is of the form '$?^{\circ}A$'. Figure 7 (top) also includes an admissible rule '$?^{\star}$' which types the syntactic sugar for bound client requests.

4 Examples

Up to here, we have presented our process language and its type system, and we have discussed the influence of asynchrony and recursion in their design and properties ensured by typing. We now present examples to further illustrate the design and expressiveness of APCP.

4.1 Milner's Typed Cyclic Scheduler

To consider a process that goes beyond the scope of PCP, here we show that our specification of Milner's cyclic scheduler from Section 2 is typable in APCP, and thus deadlock free (cf. Theorem 6). Let us recall the process definitions of the leader and followers, omitting braces '$\{\ldots\}$' for branches with one option:

$$\begin{aligned}
A_1 &:= \mu X(a_1, c_n, d_1); d_1 \triangleleft \mathsf{start} \cdot a_1 \triangleleft \mathsf{start} \cdot a_1 \triangleright \mathsf{ack}; d_1 \triangleleft \mathsf{next} \cdot c_n \triangleright \mathsf{start}; c_n \triangleright \mathsf{next}; X\langle a_1, c_n, d_1\rangle \\
A_{i+1} &:= \mu X(a_{i+1}, c_i, d_{i+1}); \; c_i \triangleright \mathsf{start}; a_{i+1} \triangleleft \mathsf{start} \cdot d_{i+1} \triangleleft \mathsf{start} \cdot a_{i+1} \triangleright \mathsf{ack}; \qquad \forall 1 \leq i < n \\
&\qquad c_i \triangleright \mathsf{next}; d_{i+1} \triangleleft \mathsf{next} \cdot X\langle a_{i+1}, c_i, d_{i+1}\rangle
\end{aligned}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash X{:}3; a_1{:}X, c_n{:}X, d_1{:}X}\ \textsc{Var}}{\vdash X{:}3; a_1{:}X, c_n{:}\&^{\rho_n}\{\mathsf{next}: X\}, d_1{:}X}\ \&}{\vdash X{:}3; a_1{:}X, c_n{:}\&^{\pi_n}\{\mathsf{start}: \&^{\rho_n}\{\mathsf{next}: X\}\}, d_1{:}X}\ \&}{\vdash X{:}3; a_1{:}X, c_n{:}\&^{\pi_n}\{\mathsf{start}: \&^{\rho_n}\{\mathsf{next}: X\}\}, d_1{:}\oplus^{\rho_1}\{\mathsf{next}: X\}}\ \oplus^\star}{\vdash X{:}3; a_1{:}\&^{\kappa_1}\{\mathsf{ack}: X\}, c_n{:}\&^{\pi_n}\{\mathsf{start}: \&^{\rho_n}\{\mathsf{next}: X\}\}, d_1{:}\oplus^{\rho_1}\{\mathsf{next}: X\}}\ \&}{\vdash X{:}3; a_1{:}\oplus^{\mathsf{o}_1}\{\mathsf{start}: \&^{\kappa_1}\{\mathsf{ack}: X\}\}, c_n{:}\&^{\pi_n}\{\mathsf{start}: \&^{\rho_n}\{\mathsf{next}: X\}\}, d_1{:}\oplus^{\rho_1}\{\mathsf{next}: X\}}\ \oplus^\star}{\begin{array}{l}\vdash X{:}3;\ a_1{:}\oplus^{\mathsf{o}_1}\{\mathsf{start}: \&^{\kappa_1}\{\mathsf{ack}: X\}\},\ c_n{:}\&^{\pi_n}\{\mathsf{start}: \&^{\rho_n}\{\mathsf{next}: X\}\},\\ \qquad d_1{:}\oplus^{\pi_1}\{\mathsf{start}: \oplus^{\rho_1}\{\mathsf{next}: X\}\}\end{array}}\ \oplus^\star}{\begin{array}{l}\vdash \emptyset;\ a_1{:}\mu X.\oplus^{\mathsf{o}_1}\{\mathsf{start}: \&^{\kappa_1}\{\mathsf{ack}: X\}\},\ c_n{:}\mu X.\&^{\pi_n}\{\mathsf{start}: \&^{\rho_n}\{\mathsf{next}: X\}\},\\ \qquad d_1{:}\mu X.\&^{\pi_1}\{\mathsf{start}: \oplus^{\rho_1}\{\mathsf{next}: X\}\}\end{array}}\ \textsc{Rec}$$

Figure 8: Typing derivation of the leader scheduler A_1 of Milner's cyclic scheduler (processes omitted).

Each process A_{i+1} for $0 \leq i < n$—thus including the leader—is typable as follows, assuming c_i is c_n for $i = 0$ (see Fig. 8 for the derivation of A_1, omitting processes from judgments):

$$\begin{array}{l}A_{i+1} \vdash \emptyset;\ a_{i+1}{:}\mu X.\oplus^{\mathsf{o}_{i+1}}\{\mathsf{start}: \&^{\kappa_{i+1}}\{\mathsf{ack}: X\}\},\ c_i{:}\mu X.\&^{\pi_i}\{\mathsf{start}: \&^{\rho_i}\{\mathsf{next}: X\}\},\\ \qquad d_{i+1}{:}\mu X.\oplus^{\pi_{i+1}}\{\mathsf{start}: \oplus^{\rho_{i+1}}\{\mathsf{next}: X\}\}\end{array}$$

Note how, for each $1 \leq i \leq n$, the types for c_i and d_i are duals.

To verify these typing derivations, we need to assign values to the priorities $\mathsf{o}_i, \kappa_i, \pi_i, \rho_i$ for each $1 \leq i \leq n$ that satisfy the necessary requirements. From the derivation of A_1 we require $\kappa_1 < \rho_1, \pi_n$. For each $1 \leq i < n$, from the derivation of A_{i+1} we require $\rho_i < \rho_{i+1}$ and $\kappa_{i+1} < \rho_i, \rho_{i+1}$ and $\pi_i < \mathsf{o}_{i+1}, \pi_{i+1}$. We can easily satisfy these requirements by assigning $\mathsf{o}_i := \kappa_i := \pi_i := i$ and $\rho_i := i+2$ for each $1 \leq i \leq n$.

Assuming that $P_i \vdash \emptyset; a_i{:}\mu X.\&^{\mathsf{o}_i}\{\mathsf{start}: \oplus^{\kappa_i}\{\mathsf{ack}: X\}\}$ for each $1 \leq i \leq n$, we have $Sched_n \vdash \emptyset;\emptyset$. Hence, it follows from Theorem 6 that $Sched_n$ is deadlock free for each $n \geq 1$.

4.2 Comparison to Padovani's Type System for Deadlock Freedom

Padovani's type system for deadlock freedom [21] simplifies a type system by Kobayashi [17]; both these works do not consider session types. Just as for Dardha and Gay's PCP [8], the priority annotations of APCP are based on similar annotations in Padovani's and Kobayashi's type systems. Here, we compare APCP to these type systems by discussing some of the examples in Padovani's work.

Ring of Processes To illustrate APCP's flexible support for recursion, we consider Padovani's ever-growing ring of processes [21, Ex. 3.8]. For the ring to continuously loop, Padovani uses self-replicating processes. Although this exact method is not possible in APCP, we can use recursion instead:

$$Ring_x^y := \mu X(x,y); (\boldsymbol{\nu} a a')(x(z); (\boldsymbol{\nu} b b')(X\langle z,b\rangle \mid X\langle b',a\rangle) \mid \overline{y}[c] \cdot a' \leftrightarrow c) \vdash \emptyset; x{:}\mu X.X \mathbin{⅋}^{\mathsf{o}} \bullet, y{:}\mu X.X \otimes^{\kappa} \bullet$$

Each iteration, this process receives a fresh endpoint from its left neighbor and sends another fresh endpoint to its right neighbor. It then spawns two copies of itself, connected to each other on a fresh channel, and one connected to the left neighbor and the other to the right neighbor. There are no priority

requirements, so we can let $\circ = \kappa$. We can then connect the initial copy of *Ring* to itself, forming a deadlock free ring of processes that doubles in size at every iteration (cf. Theorem 6):

$$
\begin{gathered}
(\boldsymbol{\nu} xy)Ring_x^y \longrightarrow^3 (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(Ring_{x_1}^{y_2} \mid Ring_{x_2}^{y_1}) \\
\longrightarrow^6 (\boldsymbol{\nu} x_1y_1)(\boldsymbol{\nu} x_2y_2)(\boldsymbol{\nu} x_3y_3)(\boldsymbol{\nu} x_4y_4)(Ring_{x_1}^{y_2} \mid Ring_{x_2}^{y_3} \mid Ring_{x_3}^{y_4} \mid Ring_{x_4}^{y_1}) \longrightarrow^{12} \ldots
\end{gathered}
$$

Blocking versus Non-blocking Padovani discusses the significance of blocking inputs versus non-blocking outputs [21, Exs. 2.2 & 3.6]. Although we can express Padovani's example in APCP with minor modifications, we can do so more directly by including replication as in Section 3.4. Consider the following processes, which are identical up to the order of input and output:

$$
Node_A := !c_A(c); c(x); c(y); \overline{x}[a] \cdot y(z); \mathbf{0} \qquad Node_B := !c_B(c); c(x); c(y); y(z); \overline{x}[a] \cdot \mathbf{0}
$$

We consider several configurations of nodes, using the syntactic sugar $\overline{x}\langle y\rangle \cdot P := \overline{x}[y'] \cdot (y \leftrightarrow y' \mid P)$:

$$
\begin{aligned}
L_1(X) &:= (\boldsymbol{\nu} c_A c_A')(\boldsymbol{\nu} c_B c_B')(Node_A \mid Node_B \mid ?\overline{c_X'}[c] \cdot (\boldsymbol{\nu} ee')(\overline{c}\langle e\rangle \cdot \overline{c}\langle e'\rangle \cdot \mathbf{0})) \\
L_2(X,Y) &:= (\boldsymbol{\nu} c_A c_A')(\boldsymbol{\nu} c_B c_B')(Node_A \mid Node_B \mid (\boldsymbol{\nu} ee')(\boldsymbol{\nu} ff') \begin{pmatrix} ?\overline{c_X'}[c] \cdot \overline{c}\langle e\rangle \cdot \overline{c}\langle f\rangle \cdot \mathbf{0} \\ \mid ?\overline{c_Y'}[c'] \cdot \overline{c'}\langle f'\rangle \cdot \overline{c'}\langle e'\rangle \cdot \mathbf{0} \end{pmatrix}) \\
L_3(X,Y,Z) &:= (\boldsymbol{\nu} c_A c_A')(\boldsymbol{\nu} c_B c_B')(Node_A \mid Node_B \mid (\boldsymbol{\nu} ee')(\boldsymbol{\nu} ff')(\boldsymbol{\nu} gg') \begin{pmatrix} ?\overline{c_X'}[c] \cdot \overline{c}\langle e\rangle \cdot \overline{c}\langle f\rangle \cdot \mathbf{0} \\ \mid ?\overline{c_Y'}[c'] \cdot \overline{c'}\langle g\rangle \cdot \overline{c'}\langle e'\rangle \cdot \mathbf{0} \\ \mid ?\overline{c_Z'}[c''] \cdot \overline{c''}\langle f'\rangle \cdot \overline{c''}\langle g'\rangle \cdot \mathbf{0} \end{pmatrix})
\end{aligned}
$$

where $X,Y,Z \in \{A,B\}$.

To illustrate the significance of APCP's asynchrony, let us consider how $L_2(A,A)$ reduces:

$$
L_2(A,A) \longrightarrow^6 (\boldsymbol{\nu} ee')(\boldsymbol{\nu} ff')(\overline{e}[a] \cdot f(z); \mathbf{0} \mid \overline{f'}[a'] \cdot e'(z'); \mathbf{0}) \longrightarrow (\boldsymbol{\nu} ff')(f(z); \mathbf{0} \mid \overline{f'}[a'] \cdot \mathbf{0}) \longrightarrow \mathbf{0}.
$$

The synchronization on e and e' is possible because the output on f' is non-blocking. It is also possible for f and f' to synchronize first, because the output on e is also non-blocking. In contrast, the reduction of $L_2(B,B)$ illustrates the blocking behavior of inputs:

$$
L_2(B,B) \longrightarrow^6 (\boldsymbol{\nu} ee')(\boldsymbol{\nu} ff')(f(z); \overline{e}[a] \cdot \mathbf{0} \mid e'(z'); \overline{f'}[a'] \cdot \mathbf{0}) \not\longrightarrow.
$$

This results in deadlock, for each node awaits a message, blocking their output to the other node.

Let us show how APCP detects (freedom of) deadlocks in each of these configurations by considering priority requirements. For $X \in \{A,B\}$, we have $Node_X \vdash \emptyset; c_X : !^{\circ}((\bullet \otimes^{\kappa_X} \bullet) \parr^{\pi_X} (\bullet \parr^{\rho_X} \bullet) \parr^{\psi_X} \bullet)$, requiring $\rho_B < \kappa_B$ and $\pi_X, \psi_X < \kappa_X, \rho_X$. In each configuration, the input endpoint of one node is connected to the output endpoint of another. Duality thus requires that $\kappa_W = \rho_{W'}$ for $W, W' \in \{X,Y,Z\}$. Hence, in any configuration, if the input endpoint of a $Node_B$ is connected the output endpoint of another $Node_B$, we require $\kappa_B = \rho_B$, violating the requirement that $\rho_B < \kappa_B$. From this we can conclude that the above configurations are deadlock free if and only if at least one of X,Y,Z is A, and at most one of them is B. This verifies that $L_2(A,A)$ contains no deadlock, while $L_2(B,B)$ does.

Note that in PCP the conditions for deadlock freedom are much stricter, as PCP's blocking outputs additionally require that $\kappa_A < \rho_A$. Hence, we also cannot connect the input of a $Node_A$ to the output of another $Node_A$. This means that $L_2(A,B)$ and $L_2(B,A)$ are the only deadlock free configurations in PCP.

5 Related Work & Conclusion

We have already discussed several related works throughout the paper [8, 10, 17, 21]. The work of Kobayashi and Laneve [18] is related to APCP in that it addresses deadlock freedom for *unbounded* process networks. Another related approach is Toninho and Yoshida's [26], which addresses deadlock freedom for cyclic process networks by generating global types from binary types. The work by Balzer *et al.* [1, 2] is also worth mentioning: it guarantees deadlock freedom for processes with shared, mutable resources by means of manifest sharing, i.e. explicitly acquiring and releasing access to resources. Finally, Pruiksma and Pfenning's session type system derived from adjoint logic [23, 24] treats asynchronous, non-blocking actions via axiomatic typing rules, similarly as we do (cf. axioms '$\otimes$' and '$\oplus$' in Figure 3); we leave a precise comparison with their approach for future work.

In this paper, we have presented APCP, a type system for deadlock freedom of cyclic process networks with asynchronous communication and recursion. We have shown that, when compared to (the synchronous) PCP [8], asynchrony in APCP significantly simplifies the management of priorities required to detect cyclic dependencies (cf. the discussion at the end of Section 4.2). We illustrated the expressivity of APCP using multiple examples, and concluded that it is comparable in expressivity to similar type systems not based on session types or logic, in particular the one by Padovani [21]. More in-depth comparisons with this and the related type systems cited above would be much desirable. Finally, in ongoing work we are applying APCP to the analysis of multiparty protocols implemented as processes [13].

Acknowledgements We are grateful to the anonymous reviewers for their careful reading of our paper and their useful feedback. We also thank Ornela Dardha for clarifying the typing rules of PCP to us.

References

[1] Stephanie Balzer & Frank Pfenning (2017): *Manifest Sharing with Session Types*. Proc. ACM Program. Lang. 1(ICFP), pp. 37:1–37:29, doi:10.1145/3110281.

[2] Stephanie Balzer, Bernardo Toninho & Frank Pfenning (2019): *Manifest Deadlock-Freedom for Shared Session Types*. In Luís Caires, editor: Programming Languages and Systems, Lecture Notes in Computer Science, Springer International Publishing, Cham, pp. 611–639, doi:10.1007/978-3-030-17184-1_22.

[3] Michele Boreale (1998): *On the Expressiveness of Internal Mobility in Name-Passing Calculi*. Theoretical Computer Science 195(2), pp. 205–226, doi:10.1016/S0304-3975(97)00220-X.

[4] Gérard Boudol (1992): *Asynchrony and the Pi-Calculus*. Research Report RR-1702, INRIA.

[5] Luís Caires (2014): *Types and Logic, Concurrency and Non-Determinism*. Technical Report MSR-TR-2014-104, In Essays for the Luca Cardelli Fest, Microsoft Research.

[6] Luís Caires & Jorge A. Pérez (2017): *Linearity, Control Effects, and Behavioral Types*. In Hongseok Yang, editor: Programming Languages and Systems, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 229–259, doi:10.1007/978-3-662-54434-1_9.

[7] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: CONCUR 2010 - Concurrency Theory, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[8] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: Foundations of Software Science and Computation Structures, Lecture Notes in Computer Science, Springer International Publishing, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[9] Ornela Dardha & Jorge A. Pérez (2015): *Comparing Deadlock-Free Session Typed Processes*. *Electronic Proceedings in Theoretical Computer Science* 190, pp. 1–15, doi:10.4204/EPTCS.190.1.

[10] Henry DeYoung, Luís Caires, Frank Pfenning & Bernardo Toninho (2012): *Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication*. In Patrick Cégielski & Arnaud Durand, editors: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, *Leibniz International Proceedings in Informatics (LIPIcs)* 16, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 228–242, doi:10.4230/LIPIcs.CSL.2012.228.

[11] Simon J. Gay, Peter Thiemann & Vasco T. Vasconcelos (2020): *Duality of Session Types: The Final Cut*. *Electronic Proceedings in Theoretical Computer Science* 314, pp. 23–33, doi:10.4204/EPTCS.314.3.

[12] Jean-Yves Girard (1987): *Linear Logic*. *Theoretical Computer Science* 50(1), pp. 1–101, doi:10.1016/0304-3975(87)90045-4.

[13] Bas van den Heuvel & Jorge A. Pérez (2021): *A Decentralized Analysis of Multiparty Protocols*. *arXiv:2101.09038 [cs]*.

[14] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR'93*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[15] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In Pierre America, editor: *ECOOP'91 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 133–147, doi:10.1007/BFb0057019.

[16] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 122–138, doi:10.1007/BFb0053567.

[17] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In Christel Baier & Holger Hermanns, editors: *CONCUR 2006 – Concurrency Theory*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 233–247, doi:10.1007/11817949_16.

[18] Naoki Kobayashi & Cosimo Laneve (2017): *Deadlock Analysis of Unbounded Process Networks*. *Information and Computation* 252, pp. 48–70, doi:10.1016/j.ic.2016.03.004.

[19] Robin Milner (1989): *Communication and Concurrency*. Prentice Hall International Series in Computer Science, Prentice Hall, New York, USA.

[20] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Information and Computation* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[21] Luca Padovani (2014): *Deadlock and Lock Freedom in the Linear π-Calculus*. In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, ACM, New York, NY, USA, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[22] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts.

[23] Klaas Pruiksma & Frank Pfenning (2019): *A Message-Passing Interpretation of Adjoint Logic*. In: *Programming Language Approaches to Concurrency- and Communication-cEntric Software (PLACES)*, *Electronic Proceedings in Theoretical Computer Science* 291, Open Publishing Association, pp. 60–79, doi:10.4204/EPTCS.291.6.

[24] Klaas Pruiksma & Frank Pfenning (2021): *A Message-Passing Interpretation of Adjoint Logic*. *Journal of Logical and Algebraic Methods in Programming* 120(100637), doi:10.1016/j.jlamp.2020.100637.

[25] Bernardo Toninho, Luis Caires & Frank Pfenning (2014): *Corecursion and Non-Divergence in Session-Typed Processes*. In Matteo Maffei & Emilio Tuosto, editors: *Trustworthy Global Computing*, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, pp. 159–175, doi:10.1007/978-3-662-45917-1_11.

[26] Bernardo Toninho & Nobuko Yoshida (2018): *Interconnectability of Session-Based Logical Processes*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40(4), p. 17, doi:10.1145/3242173.

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

[28] Philip Wadler (2012): *Propositions As Sessions*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, ACM, New York, NY, USA, pp. 273–286, doi:10.1145/2364527.2364568.