ICE 2021 Pre-Proceedings

Julien Lange Anastasia Mavridou Larisa Safina Alceste Scalas

15th June 2021

This document contains the *informal* pre-proceedings of the 14th Interaction and Concurrency Experience (ICE 2021). The post-proceedings will be published on EPTCS.

Contents

Multiparty Session Programming with Global Protocol Combinators (Oral Contribution)

Keigo Imai
keigoi@gifu-u.ac.jp
Gifu Gifu University
Japan

Rumyana Neykova
Rumyana.Neykova@brunel.ac.uk
London Brunel University London
UK

Nobuko Yoshida
n.yoshida@imperial.ac.uk
London Imperial College London
UK

Shoji Yuen
yuen@i.nagoya-u.ac.jp
Nagoya Nagoya University
Japan

Multiparty Session Types. Multiparty Session Types (MPST) [1–3] is a theoretical framework that stipulates how to write, verify and ensure correct implementations of communication protocols. The methodology of programming with MPST (depicted in Fig. 1) starts from a communication protocol (a *global type*) which specifies the behaviour of a system of interacting processes. The local behaviour (a *local type*) for each endpoint process is then algorithmically *projected* from the protocol. Finally, each endpoint process is implemented in an endpoint host language and type-checked against its respective local type by a session typing system. The guarantee of session types is that a system of well-typed endpoint processes *does not go wrong*, i.e it does not exhibit communication errors such as reception errors, orphan messages or deadlocks, and satisfies session fidelity, i.e. the local behaviour of each process follows the global specification.

Our approach. This talk presents `ocaml-mpst`, a library for programming MPST protocols in OCaml which allows to specify, verify and implement MPST protocols in a single language, OCaml. Specifically, we introduce *global combinators*, a statically typed, embedded DSL (EDSL) for writing global types. Thus, an *unsafe* global protocol can be detected as a *type error* by OCaml typecher.

Our approach consists of (1) the encoding of local types using variant and record (sub)typing (which are pervasive in functional programming languages), and (2) the formulation of global protocols as a *term-level* (not *type*-level) object, with a set of *typing rules*, which is designed to follow the projection algorithm, making the global combinators having the projected local behaviour in their types.

The benefits of `ocaml-mpst` are that it is (1) *lightweight* – it does not depend on any external code-generation mechanism, verification of global protocols is reduced to typability of global combinators; (2) *fully-static* – our embedding integrates with recent techniques for static checking of linearly typed programming in OCaml [4, 5], and (3) *expressive* – we can type strictly more processes than [6].

In our talk, we introduce the design of global combinators (Fig. 2) and show our key finding that the *merging* of local behaviours coincides with the existence of least upper bound w.r.t. subtyping relation (Fig. 3), on top of that the typing of global combinators is implemented. We also briefly show the *channel vector semantics* of global combinators, which gives the run-time entity of the multiparty channels using Concurrent ML's standard `Event` module.

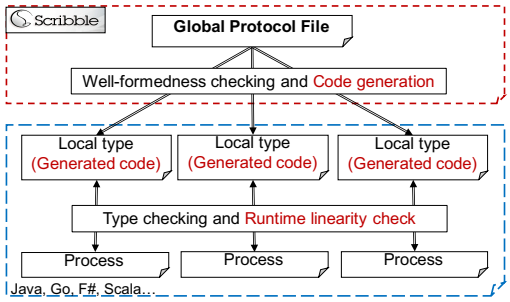


Figure 1. State-of-the-art MPST implementations

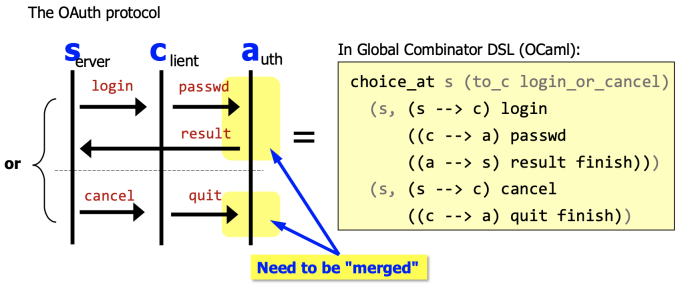


Figure 2. Merging in MPST

References

[1] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *Formal Methods for Multicore Programming (LNCS, Vol. 9104)*. Springer, 146–178. https://doi.org/10.1007/978-3-319-18941-3_4

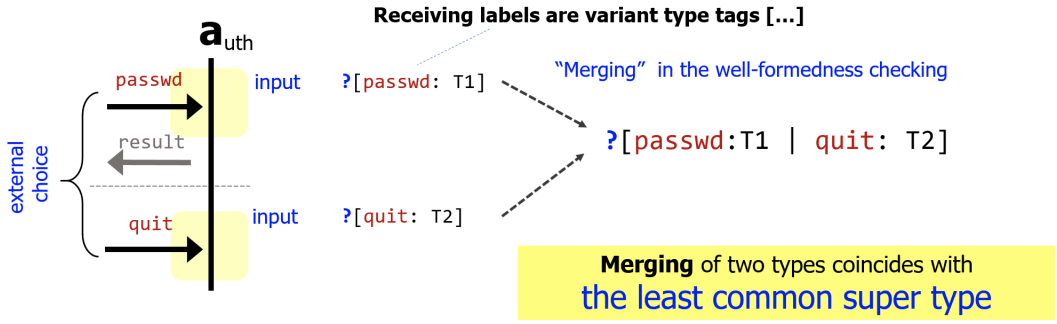


Figure 3. Merging of local types in `ocaml-mpst`

[2] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL'08*. ACM, 273–284.

[3] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. https://doi.org/10.1145/2827695

[4] Keigo Imai and Jacques Garrigue. 2019. Lightweight Linearly-typed Programming with Lenses and Monads. *Journal of Information Processing* 27 (2019), 431–444. https://doi.org/10.2197/ipsjjip.27.431

[5] Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2018. Session-ocaml: a Session-based Library with Polarities and Lenses. *Sci. Comput. Program.* 172 (2018), 135–159. https://doi.org/10.1016/j.scico.2018.08.005

[6] Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP (LIPIcs, Vol. 56)*. 21:1–21:28. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

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).

Submitted to:
ICE 2021

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*.

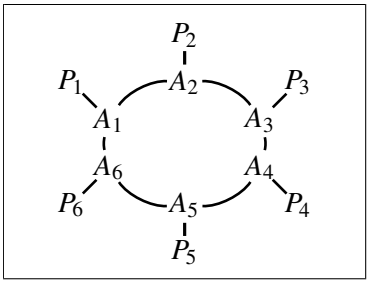


Figure 1: Milner's cyclic scheduler with 6 workers.
Lines denote channels connecting processes.

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:

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

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

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:

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

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:

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

Structural congruence:

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

Reduction:

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

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

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 $\beta_{\otimes\parr}$ 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 $\kappa_{\parr}$ (resp. $\kappa_{\&}$) pulls an input (resp. a branching) prefix on free channels out of enclosing restrictions. Rules $\rightarrow_{\equiv}$, $\rightarrow_{\nu}$, and $\rightarrow_{|}$ 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 $\circ, \kappa, \pi, \rho, \ldots$ 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 $\circ \in \mathbb{N} \cup \{\omega\}$.*

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

An endpoint of type '$A \otimes^{\circ} B$' (resp. '$A \parr^{\circ} B$') first outputs (resp. inputs) an endpoint of type A and then behaves as B. An endpoint of type '$\&^{\circ}\{i : A_i\}_{i \in I}$' offers a choice: after receiving a label $i \in I$, the endpoint behaves as A_i. An endpoint of type '$\oplus^{\circ}\{i : A_i\}_{i \in I}$' 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 '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, '*pr(A)*' denotes its* priority*:*

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

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 '$\parr$' (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:*

$$\uparrow^t (A \otimes^{\circ} B) := (\uparrow^t A) \otimes^{\circ+t} (\uparrow^t B) \qquad \uparrow^t (\oplus^{\circ}\{i : A_i\}_{i \in I}) := \oplus^{\circ+t}\{i : \uparrow^t A_i\}_{i \in I} \qquad \uparrow^t \bullet := \bullet$$
$$\uparrow^t (A \parr^{\circ} B) := (\uparrow^t A) \parr^{\circ+t} (\uparrow^t B) \qquad \uparrow^t (\&^{\circ}\{i : A_i\}_{i \in I}) := \&^{\circ+t}\{i : \uparrow^t A_i\}_{i \in I}$$
$$\uparrow^t (\mu X.A) := \mu X.\uparrow^t (A) \qquad \uparrow^t X := X$$

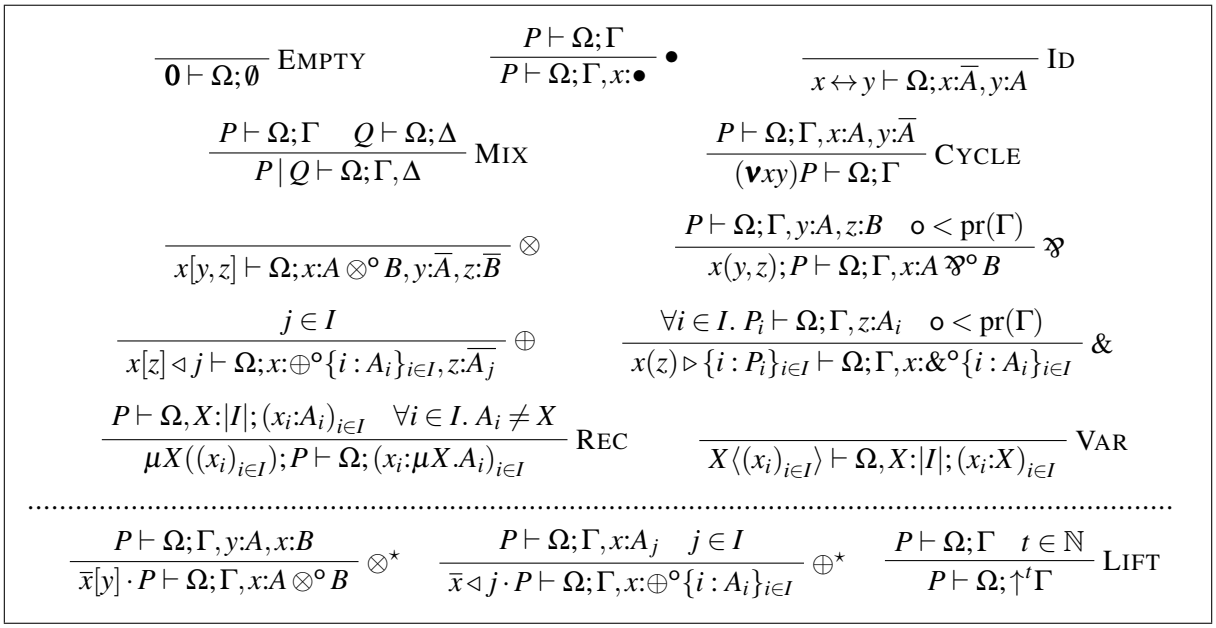


Figure 3: The typing rules of APCP (top) and admissible rules (bottom).

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'

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

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

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

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 '⅋' 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. 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

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

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

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

$$\longrightarrow$$

$$\frac{\dfrac{\dfrac{P \vdash \Gamma, \Gamma', y{:}A, z{:}B \quad Q \vdash \Delta, \Delta'}{(\boldsymbol{\nu} \tilde{v}\tilde{w})(P \mid Q) \vdash \Gamma, \Delta, y{:}A, z{:}B}\ \text{MIX} + \text{CYCLE}^*}{(\boldsymbol{\nu} \tilde{v}\tilde{w})(P \mid Q) \vdash \uparrow^{\circ+1}\Gamma, \uparrow^{\circ+1}\Delta, y{:}\uparrow^{\circ+1}A, z{:}\uparrow^{\circ+1}B}\ \text{LIFT} \quad \circ < \mathrm{pr}(\uparrow^{\circ+1}\Gamma, \uparrow^{\circ+1}\Delta)}{x(y,z).(\boldsymbol{\nu} \tilde{v}\tilde{w})(P \mid Q) \vdash \uparrow^{\circ+1}\Gamma, \uparrow^{\circ+1}\Delta, x{:}(\uparrow^{\circ+1}A) \parr^{\circ} (\uparrow^{\circ+1}B)}\parr$$

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

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.

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.
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):*

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

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 $P^\star := P$.

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

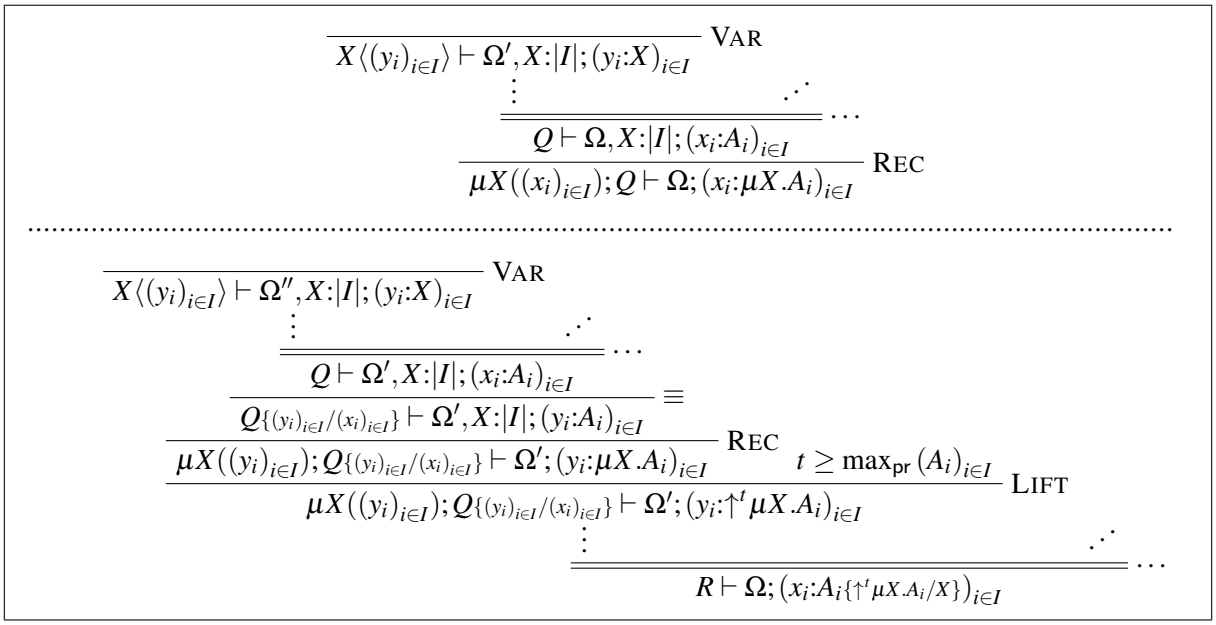


Figure 6: Typing recursion before (top) and after (bottom) unfolding (cf. Lemma 3).

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* 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) $A_j = B \parr^{\circ} C$ or (b) $A_j = \&^{\circ}\{l : B_l\}_{l\in L}$.
  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) $\overline{A_j} = \overline{B} \otimes^{\circ} \overline{C}$ or (in case b) $\overline{A_j} = \oplus^{\circ}\{l : \overline{B_l}\}_{l\in L}$, 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. $P \equiv (\boldsymbol{\nu}\tilde{u}\tilde{v})(\boldsymbol{\nu} x x')(x(y,z);P' \mid Q)$ 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, •, 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 '•' with '$\mathbf{1}^\circ$' and '$\bot^\circ$', associated to empty 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 '•' 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

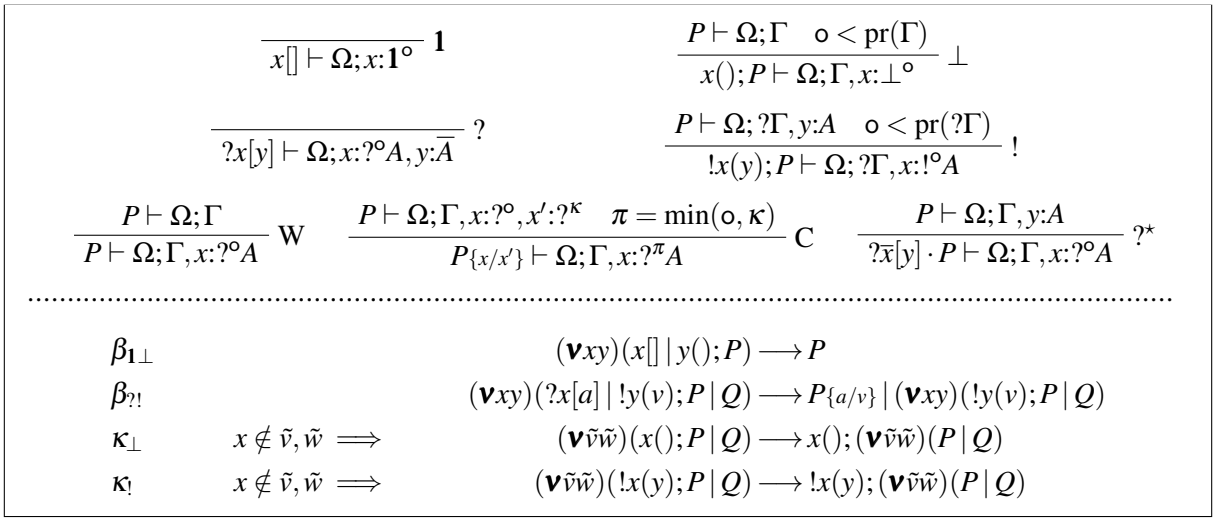


Figure 7: Typing rules for explicit closing and replicated servers.

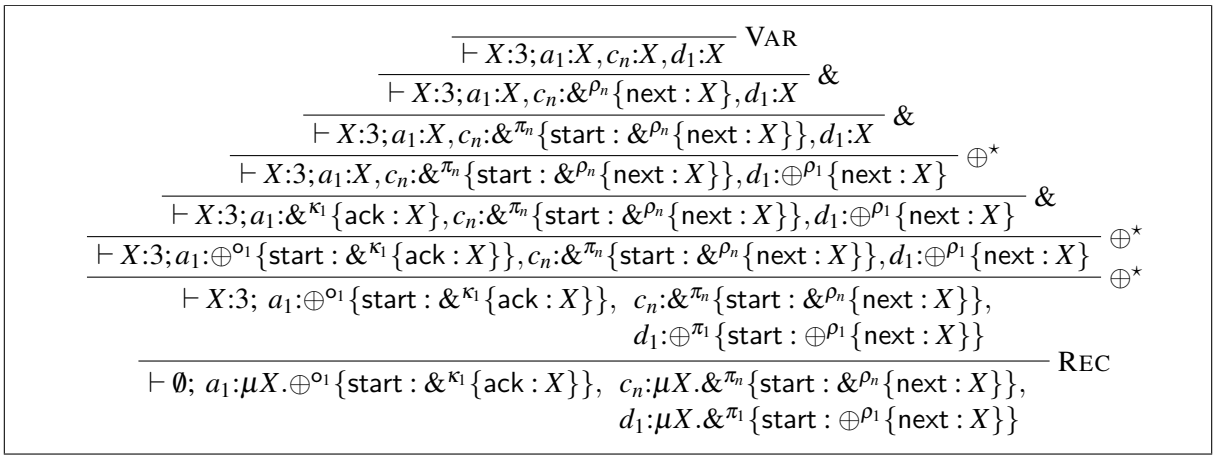


Figure 8: Typing derivation of the leader scheduler A_1 of Milner's cyclic scheduler (processes omitted).

servers, respectively, and the latter two are for connecting to a server without requests and for multiple requests, respectively. In rule '!', notation '?Γ' 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:

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

Each process A_{i+1} for $0 \le i < n$—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):

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

Note how, for each $1 \le i \le n$, 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 $1 \le i \le n$ that satisfy the necessary requirements. From the derivation of A_1 we require $\kappa_1 < \rho_1, \pi_n$. For each $1 \le i < n$, 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 $1 \le i \le n$.

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

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:

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

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 $\mathsf{o} = \kappa$. 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):

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

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 $\overline{x}\langle y\rangle\cdot P := \overline{x}[y']\cdot(y\leftrightarrow y' \mid P)$:

$$L_1(X) := (\boldsymbol{\nu} c_A c_A')(\boldsymbol{\nu} c_B c_B')(Node_A \mid Node_B \mid ?\overline{c_X'}[c]\cdot(\boldsymbol{\nu} ee')(\overline{c}\langle e\rangle\cdot\overline{c}\langle e'\rangle\cdot\mathbf{0}))$$

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

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

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.

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.

Decomposing Monolithic Processes in a Process Algebra with Multi-actions

Compositional minimisation can be an effective technique to deal with the state space explosion problem. This technique considers a parallel composition of multiple sequential processes. In its simplest form, each sequential process is replaced by a minimal process such that the composition of these minimal processes has a smaller state space that is behaviourally equivalent to the state space of the original parallel composition. However, for a monolithic process, *i.e.*, a process without parallel composition, this technique is not applicable. Therefore, we present a technique that decomposes a monolithic process with data parameters into multiple processes where each process defines behaviour for a subset of the parameters of the monolithic process. We prove that the composition of these processes is strongly bisimilar to the monolithic process under a suitable synchronisation context. Moreover, we prove that state invariants can be used to further improve the effectiveness of the decomposition. Finally, we apply the decomposition technique to several specifications.

1 Introduction

The mCRL2 language [11] is a process algebra with multi-actions that can be used to specify the behaviour of communicating processes with data parameters. The corresponding mCRL2 toolset [4] translates the process specification, which includes (interleaving) parallel composition and operators to describe process communication, to a strongly bisimilar (non-deterministic sequential) monolithic process. Translating a complicated process specification into a simpler normal form, in this case the monolithic process, has several advantages. First of all, the design and implementation of state space exploration algorithms can be greatly simplified. Furthermore, the design of effective static analysis techniques on the global behaviour of the specification is also easier. One example is a static analysis to detect live variables as presented in [16]. However, the static analysis techniques available at the moment are not always strong enough to mitigate the state space explosion problem for this monolithic process even though its state space can often be minimised with respect to some equivalence relation after state space exploration.

In the literature there are several promising techniques to deal with the state space explosion problem. One of these techniques is *compositional minimisation* which can be used to obtain a smaller state space that is behaviourally equivalent to the state space of the original specification. The idea is that state space explosion often occurs due to all the possible interleaving of several processes in a parallel composition. In compositional minimisation, the state space of each sequential process (referred to as a component) is replaced by a (hopefully smaller) state space such that their composition is equivalent to the original specification [20, 19]. This means that properties of the specification can be checked on the composition.

This basic approach is not always useful, because the size of the state spaces belonging to individual components summed together might exceed the size of the original state space [7]. In particular, the state space can become infinitely large for components that rely on synchronisation to bound their behaviour. This can be avoided by specifying *interface constraints* (also known as *environmental constraints* or *context constraints*) leading to a *semantic compositional minimisation* [9, 5]. Furthermore, the order in which intermediate components are explored, minimised and subsequently composed heavily influences

To appear in EPTCS.

the size of the intermediate state spaces. There are heuristics for these problems that can be very effective in practice, as shown by the CADP [6] (Construction and Analysis of Distributed Processes) toolset.

Unfortunately, in our context where parallel composition is removed by a translation step the aforementioned compositional techniques, which rely on the user-defined parallel composition and the (sequential) processes, are not applicable. In this paper, we define a decomposition technique (which we refer to as a *cleave*) of a monolithic process based on a partitioning of its data parameters that yields two components. This way we have the monolithic process available for static analysis (with corresponding transformations) and we can use compositional minimisation. We show that under a suitable synchronisation context the composition of the two components is *strongly bisimilar* to the monolithic process. The technique can be applied recursively to obtain a number of components to which composition minimisation can be applied. We also perform a case study to evaluate the decomposition technique in practice.

The advantages of decomposing the monolithic process are the same as for compositional minimisation. First of all, by minimising the state spaces of intermediate components the composed state space can be immediately smaller than the state space obtained by exploring the monolithic process. Furthermore, state space exploration relies on the evaluation of data expressions of the higher-level specification language and that can be costly, whereas the composition of the state spaces of the components can be computed without evaluating expressions. An advantage of the decomposition technique over compositional minimisation is that constraints resulting from the synchronisation of these processes can be used when deriving the components. These constraints can be further improved by using so-called state invariants, as we also demonstrate in this paper. Finally, the components resulting from the decomposition are not limited to the user-defined processes present in the specification, which could yield a more optimal composition. Indeed, the case studies on which we report support both observations.

Related Work. Several different techniques are related to this type of decomposition. Most notably, the work on decomposing Petri nets into a set of automata [2] also aims to speed up state space exploration by means of decomposition. The work on functional decomposition [3] describes a technique to decompose a specification based on a partitioning of the action labels instead of a partitioning of the data parameters. In [13] it was shown how this type of decomposition can be achieved for mCRL2 processes. Furthermore, a decomposition technique was used in [10] to improve the efficiency of equivalence checking. However, that work considers processes that were already in a parallel composition and further decomposes them based on the actions that occur in each component.

Outline. In Section 2 the syntax and semantics of the considered process algebra are defined. The decomposition problem is defined in Section 3 and the cleave technique is presented in Section 4. In Section 5 the cleave technique is improved with state invariants. In Section 6 the implementation is described shortly and a case study is presented in Section 7 to illustrate the effectiveness of the decomposition technique in practice. Finally, a conclusion and future work is presented in Section 8.

2 Preliminaries

We assume the existence of an abstract data theory that describes data sorts. Each sort D has an associated non-empty semantic domain denoted by $\mathbb{D}$. The existence of sorts *Bool* and *Nat* with their associated Boolean ($\mathbb{B}$) and natural number ($\mathbb{N}$) semantic domains respectively, with standard operators is assumed. Furthermore, we assume the existence of an infinite set of *sorted variables*. We use $e : D$ to indicate that e is an expression (or variable) of sort D. The set of free variables of an expression e is denoted $\mathsf{FV}(e)$,

and a variable that is not free is called *bound*. An expression e is *closed* iff $\mathsf{FV}(e) = \emptyset$. A substitution σ is a total function from variables to closed data expressions of their corresponding sort. We use $\sigma(e)$ to denote the syntactic replacement of variables in expression e by their substituted expression.

An *interpretation* function, denoted by $[\![\ldots]\!]$, maps syntactic objects to values within their corresponding semantic domain. We assume that $[\![e]\!]$ for closed expressions e is already defined. Semantic objects are typeset in *boldface* to differentiate them from syntax, *e.g.*, the semantics of expression $1+1$ is $\mathbf{2}$. We denote *data equivalence* by $e \approx f$, which is true iff $[\![e]\!] = [\![f]\!]$; for other operators we use the same symbol in both syntactic and semantic domains. We adopt the usual principle of substitutivity; *i.e.*, for all variables x, expressions e and closed expressions g and h it holds that if $g \approx h$ then $[x \leftarrow g](e) \approx [x \leftarrow h](e)$.

We denote a *vector* of length $n+1$ by $\vec{d} = \langle d_0, \ldots, d_n \rangle$. Two vectors are equivalent, denoted by $\langle d_0, \ldots, d_n \rangle \approx \langle e_0, \ldots, e_n \rangle$, iff their elements are *pairwise equivalent*, *i.e.*, $d_i \approx e_i$ for all $0 \leq i \leq n$. Given a vector $\langle d_0, \ldots, d_n \rangle$ and a subset $I \subseteq \mathbb{N}$, we define the *projection*, denoted by $\langle d_0, \ldots, d_n \rangle_{|I}$, as the vector $\langle d_{i_0}, \ldots, d_{i_l} \rangle$ for the largest $l \in \mathbb{N}$ such that $i_0 < i_1 < \ldots < i_l \leq n$ and $i_k \in I$ for $0 \leq k \leq l$. We write $\vec{d} : \vec{D}$ for a vector of $n+1$ variables $d_0 : D_0, \ldots, d_n : D_n$ and denote the projection for a subset of indices $I \subseteq \mathbb{N}$ by $\vec{d}_{|I} : \vec{D}_{|I}$. Finally, we define $\mathsf{Vars}(\vec{d}) = \{d_0, \ldots, d_n\}$.

A *multi-set* over a set A is a total function $m : A \to \mathbb{N}$; we refer to $m(a)$ as the *multiplicity* of a and we write $\wr \ldots \int$ for a multi-set where the multiplicity of each element is either written next to it or omitted when it is one. For instance, $\wr a : 2, b \int$ has elements a and b with multiplicity two and one respectively, and all other elements have multiplicity zero. For multi-sets $m, m' : A \to \mathbb{N}$, we write $m \subseteq m'$ iff $m(a) \leq m'(a)$ for all $a \in A$. Multi-sets $m + m'$ and $m - m'$ are defined pointwise: $(m + m')(a) = m(a) + m'(a)$ and $(m - m')(a) = \max(m(a) - m'(a), 0)$ for all $a \in A$.

2.1 Labelled Transition Systems

Let Λ be the set of (sorted) action *labels*. We use D_a to indicate the sort of action label $a \in \Lambda$. The set of all multi-sets over $\{a(\mathbf{e}) \mid a \in \Lambda, \mathbf{e} \in \mathbb{D}_a\}$ is denoted Ω. Note that $\mathbb{D}_a$ is the semantic domain of D_a. In examples we typically omit the expression and parentheses whenever D_a consists of a single element.

Definition 2.1. A labelled transition system with multi-actions, abbreviated LTS, is a tuple $\mathscr{L} = (S, s_0, Act, \to)$ where S is a set of states; $s_0 \in S$ is an initial state; $Act \subseteq \Omega$ and $\to \subseteq S \times Act \times S$ is a labelled transition relation.

We typically use ω to denote an element of Act and we write $s \xrightarrow{\omega} t$ whenever $(s, \omega, t) \in \to$. As usual, a finite LTS can be depicted as an edge-labelled directed graph, where vertices represent states, the labelled edges represent the transitions, and a dangling arrow indicates the initial state.

We recall the well-known strong bisimulation equivalence relation on LTSs [14].

Definition 2.2. Let $\mathscr{L}_i = (S_i, s_i, Act_i, \to_i)$ for $i \in \{1, 2\}$ be two LTSs. A binary relation $R \subseteq S_1 \times S_2$ is a *(strong) bisimulation relation* iff for all $s\,R\,t$:

- if $s \xrightarrow{\omega}_1 s'$ then there is a state $t' \in S_2$ such that $t \xrightarrow{\omega}_2 t'$ and $s'\,R\,t'$, and
- if $t \xrightarrow{\omega}_2 t'$ then there is a state $s' \in S_1$ such that $s \xrightarrow{\omega}_1 s'$ and $s'\,R\,t'$.

States s and t are *bisimilar*, denoted $s \leftrightarroweq t$, iff $s\,R\,t$ for a bisimulation relation R. We write $\mathscr{L}_1 \leftrightarroweq \mathscr{L}_2$ iff $s_1 \leftrightarroweq s_2$ and say $\mathscr{L}_1$ and $\mathscr{L}_2$ are bisimilar.

2.2 Linear Process Equations

We draw inspiration from the process algebra mCRL2 [11], which contains multi-actions, to describe the elements of an LTS; similar concepts and constructs may appear in other shapes elsewhere.

Definition 2.3. *Multi-actions* are defined as follows:

$$\alpha ::= \tau \mid a(e) \mid \alpha|\alpha$$

Constant τ represents the empty multi-action and $a \in \Lambda$ is an action label with an expression e of sort D_a. The semantics of a multi-action α for any substitution σ, denoted by $[\![\alpha]\!]_\sigma$, is an element of Ω and defined inductively as follows: $[\![\tau]\!]_\sigma = \wr\int$, $[\![a(e)]\!]_\sigma = \wr a([\![\sigma(e)]\!])\int$ and $[\![\alpha|\beta]\!]_\sigma = [\![\alpha]\!]_\sigma + [\![\beta]\!]_\sigma$. If α is a closed expression then the substitution can be omitted.

The states and transitions of an LTS are described by means of a monolithic process called a *linear process equation*, which consists of a number of *condition-action-effect* statements, referred to as *summands*. Each summand symbolically represents a partial transition relation between the current and the next state for a multi-set of action labels. Let *PN* be a set of process *names*.

Definition 2.4. A *linear process equation* (LPE) is an equation of the form:

$$P(d:D) = \sum_{e_0:E_0} c_0 \to \alpha_0 \,.\, P(g_0) + \ \ldots \ + \sum_{e_n:E_n} c_n \to \alpha_n \,.\, P(g_n)$$

Where $P \in PN$ is the process *name*, d is the process parameter, and each:

- E_i is a sort ranged over by *sum* variable e_i (where $e_i \neq d$),
- c_i is the *enabling condition*, a boolean expression so that $\mathsf{FV}(c_i) \subseteq \{d, e_i\}$,
- α_i is a multi-action τ or $a_i^1(f_i^1)|\ldots|a_i^{n_i}(f_i^{n_i})$ such that each $a_i^k \in \Lambda$ and f_i^k is an expression of sort $D_{a_i^k}$ such that $\mathsf{FV}(f_i^k) \subseteq \{d, e_i\}$,
- g_i is an *update* expression of sort D, satisfying $\mathsf{FV}(g_i) \subseteq \{d, e_i\}$.

The $+$-operator denotes a non-deterministic choice among the summands of the LPE; the $\sum$-operator describes a non-deterministic choice among the possible values of the associated sum variable bound by the $\sum$-operator. We omit the $\sum$-operator when the sum variable does not occur freely within the condition, action and update expressions. We use $\mathop{+}_{i\in I}$ for a finite set of *indices* $I \subseteq \mathbb{N}$ as a shorthand for a number of summands.

We often consider LPEs where the parameter sort D represents a *vector*; in that case we write $d_0 : D_0, \ldots, d_n : D_n$ to indicate that there are $n+1$ parameters where each d_i has sort D_i. Similarly, we also generalise the action sorts and the sum operator in LPEs, where we permit ourselves to write $a(f_0, \ldots, f_k)$ and $\sum_{e_0:E_0,\ldots,e_l:E_l}$, respectively.

The *operational semantics* of an LPE are defined by a mapping to an LTS. Let P be the set of symbols $P(\iota)$ such that $P(d:D) = \phi_P$, for any $P \in PN$, is an LPE and ι is a closed expression of sort D.

Definition 2.5. Let $P(d:D) = \mathop{+}_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(g_i)$ be an LPE and let $\iota : D$ be a closed expression. The semantics of $P(\iota)$, denoted by $[\![P(\iota)]\!]$, is the LTS $(\mathsf{P}, P(\iota), \Omega, \to)$ where $\to$ is defined as follows: for all indices $i \in I$, closed expressions $\iota' : D$ and substitutions σ such that $\sigma(d) = \iota'$ there is a transition $P(\iota') \xrightarrow{[\![\sigma(\alpha_i)]\!]} P(\sigma(g_i))$ iff $[\![\sigma(c_i)]\!] = \mathbf{true}$.

For a given LPE, we refer to the reachable part of the LTS, induced by the LPE, as the *state space*. Note that in the interpretation of an LPE a syntactic substitution is applied to the update expressions to define the reached state. This means that different closed syntactic expressions which correspond to the same semantic object, *e.g.*, $1+1$ and 2 for our assumed sort *Nat*, result in different states. As stated by the lemma below, such states are always bisimilar.

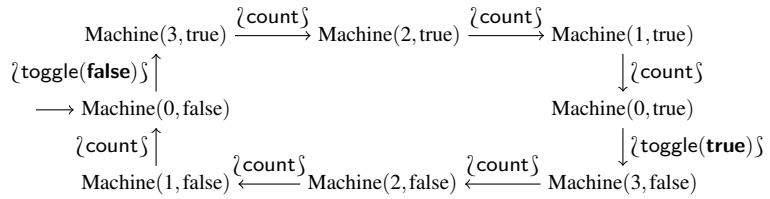


Figure 1: Example LTS for the behaviour of a machine.

Lemma 2.6. Given an LPE $P(d:D) = \phi_P$. For all closed expressions $e, e' : D$ such that $[\![e \approx e']\!] = \mathbf{true}$ we have $[\![P(e)]\!] \leftrightarrows [\![P(e')]\!]$.

For any given state space we can therefore consider a *representative* state space where for each state a unique closed expression is chosen that is data equivalent. In examples we always consider the representative state space.

Example 2.7. Consider the LPE below which models a machine that can be toggled with a delay of three counts. Whenever the counter reaches zero, *i.e.*, n is zero, it can be toggled again after which it counts down three times, *etcetera*.

$$\begin{aligned}\mathrm{Machine}(n:Nat, s:Bool) = {} & (n > 0) \to \mathsf{count}\,.\,\mathrm{Machine}(n-1, s) \\ & + (n \approx 0) \to \mathsf{toggle}(s)\,.\,\mathrm{Machine}(3, \neg s)\end{aligned}$$

A representative state space of the machine that is initially off, defined by $[\![\mathrm{Machine}(0, \mathsf{false})]\!]$, is shown in Figure 1.

2.3 A Process Algebra of Communicating Linear Process Equations

We define a minimal language to express parallelism and interaction of LPEs; the operators are taken from mCRL2 [11] and similar-styled process algebras. Let Comm be the set of *communication* expressions of the form $a_0|\ldots|a_n \to c$ where $a_0, \ldots, a_n, c \in \Lambda$ are action labels.

Definition 2.8. The process algebra is defined as follows:

$$S ::= \Gamma_C(S) \mid \nabla_A(S) \mid \tau_H(S) \mid S \parallel S \mid P(\iota)$$

Here, $A \subseteq 2^{\Lambda \to \mathbb{N}}$ is a non-empty finite set of finite multi-sets of action labels, $H \subseteq \Lambda$ is a non-empty finite set of action labels and $C \subseteq \mathsf{Comm}$ is a finite set of *communications*. Finally, we have $P(\iota) \in \mathsf{P}$.

The set S contains all expressions of the process algebra. Operator Γ_C describes *communication*, ∇_A *action allowing*, τ_H *action hiding* and $\parallel$ *parallel composition*; the elementary objects are the processes, defined as LPEs.

The operational semantics of expressions in S are defined in Definition 2.11. We first introduce three auxiliary functions on Ω that are used in the semantics.

Definition 2.9. Given $\omega \in \Omega$ we define γ_C, where $C \subseteq \mathsf{Comm}$, as follows:

$$\begin{aligned}\gamma_\emptyset(\omega) &= \omega \\ \gamma_C(\omega) &= \gamma_{C \setminus C_1}(\gamma_{C_1}(\omega)) \text{ for } C_1 \subset C \\ \gamma_{\{a_0|\ldots|a_n \to c\}}(\omega) &= \begin{cases} \{\!|c(\mathbf{d})|\!\} + \gamma_{\{a_0|\ldots|a_n \to c\}}(\omega - \{\!|a_0(\mathbf{d}), \ldots, a_n(\mathbf{d})|\!\}) \\ \quad \text{if } \{\!|a_0(\mathbf{d}), \ldots, a_n(\mathbf{d})|\!\} \subseteq \omega \\ \omega \quad \text{otherwise} \end{cases}\end{aligned}$$

For γ_C to be well-defined we require that the left-hand sides of the communications do not share labels. Furthermore, the action label on the right-hand side must not occur in any *other* left-hand side. For example $\gamma_{\{a|b\to c\}}(a|d|b) = c|d$, but $\gamma_{\{a|b\to c,a|d\to c\}}(a|d|b)$ and $\gamma_{\{a|b\to c,c\to d\}}(a|d|b)$ are not allowed.

Definition 2.10. Let $\omega \in \Omega$, $H \subseteq \Lambda$ and $\omega \in \Omega$. We define $\theta_H(\omega)$ as the multi-set ω' defined as:

$$\omega'(a(\mathbf{d})) = \begin{cases} 0 & \text{if } a \in H \\ \omega(a(\mathbf{d})) & \text{otherwise} \end{cases}$$

Finally, given a multi-action α we define $\underline{\alpha}$ to obtain the multi-set of action labels, *e.g.*, $\underline{a(3)|b(5)} = \{\!|a,b|\!\}$. Formally, $\underline{a(e)} = \{\!|a|\!\}$, $\underline{\tau} = \{\!|\,|\!\}$ and $\underline{\alpha|\beta} = \underline{\alpha} + \underline{\beta}$. We define $\underline{\omega}$ for $\omega \in \Omega$ in a similar way.

Definition 2.11. The operational semantics of an expression Q of S, denoted $[\![Q]\!]$, are defined by the corresponding LTS $(\mathsf{S}, Q, \Omega, \to)$ with its transition relation defined by the rules below and the transition relation given in Definition 2.5 for each expression in P. For any $\omega, \omega' \in \Omega$ and P, P', Q, Q' expressions of S:

$$\textsc{Com}\ \frac{P \xrightarrow{\omega} P' \qquad C \subseteq \mathsf{Comm}}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega)} \Gamma_C(P')} \qquad \textsc{Allow}\ \frac{P \xrightarrow{\omega} P' \qquad A \subseteq 2^{\Lambda\to\mathbb{N}} \qquad \underline{\omega} \in A}{\nabla_A(P) \xrightarrow{\omega} \nabla_A(P')}$$

$$\textsc{Hide}\ \frac{P \xrightarrow{\omega} P' \qquad H \subseteq \Lambda}{\tau_H(P) \xrightarrow{\theta_H(\omega)} \tau_H(P')} \qquad \textsc{Par}\ \frac{P \xrightarrow{\omega} P' \quad Q \xrightarrow{\omega'} Q'}{P \parallel Q \xrightarrow{\omega+\omega'} P' \parallel Q'}$$

$$\textsc{ParR}\ \frac{Q \xrightarrow{\omega} Q'}{P \parallel Q \xrightarrow{\omega} P \parallel Q'} \qquad \textsc{ParL}\ \frac{P \xrightarrow{\omega} P'}{P \parallel Q \xrightarrow{\omega} P' \parallel Q}$$

3 The Decomposition Problem

The state space of a monolithical LPE may grow quite large and generating that state space may either take too long or require too much memory. We are therefore interested in decomposing an LPE into two or more LPEs, where the latter are referred to as *components*, such that the state spaces of the resulting components are smaller than that of the original state space. Such a decomposition is considered *valid* iff the original state space is strongly bisimilar to the state space of these components when combined under a suitable context. We formalise this problem as follows.

Definition 3.1. Let $P(\vec{d} : \vec{D}) = \phi$ be an LPE and $\vec{\iota} : \vec{D}$ a closed expression. The LPEs $P_0(\vec{d}_{|I_0} : \vec{D}_{|I_0}) = \phi_0$ to $P_n(\vec{d}_{|I_n} : \vec{D}_{|I_n}) = \phi_n$, for indices $I_0, \ldots, I_n \subseteq \mathbb{N}$, are a valid *decomposition* of P and $\vec{\iota}$ iff there is a context C such that:

$$[\![P(\vec{\iota})]\!] \leftrightarrows [\![\mathsf{C}[P_0(\vec{\iota}_{|I_0}) \parallel \ldots \parallel P_n(\vec{\iota}_{|I_n})]]\!]$$

Where $\mathsf{C}[P_0(\vec{\iota}_{|I_0}) \parallel \ldots \parallel P_n(\vec{\iota}_{|I_n})]$ is an expression in S. We refer to the expression $\mathsf{C}[P_0(\vec{\iota}_{|I_0}) \parallel \ldots \parallel P_n(\vec{\iota}_{|I_n})]$ as the *composition*.

In the next sections, we will show that a suitable context C can be constructed using the operators from S, and we define a decomposition technique that results in exactly two components (a *cleave*). The technique can, in principle, be applied recursively to the smaller components. The primary benefit of

a valid decomposition is that a state space that is equivalent to the original state space can be obtained as follows. First, the state space of each component is derived separately. The composition can then be derived from the component state spaces, exploiting the rules of the operational semantics. The component state spaces can be minimised modulo bisimilarity, which is a congruence with respect to the operators of S before deriving the results of the composition expression. The composition resulting from these minimised components can be considerably smaller than the original state space, because also the original state space can often be reduced considerably modulo strong bisimilarity after generation. This process is referred to as *compositional minimisation*.

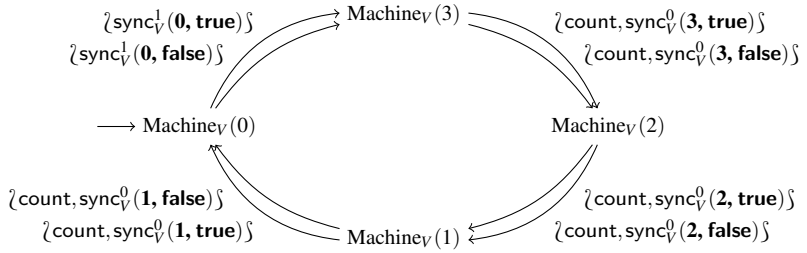
4 A Solution to the Decomposition Problem

A basic observation that we exploit in our solution to the decomposition problem is that when hiding label c in a multi-action $\alpha|c$, we are left with multi-action α, provided that c does not occur in α. When the multi-action α is an event that is possible in a monolithic LPE and the label c is the result of a communication of two components, we can effectively exchange information between multiple components, without this information becoming visible externally. The example below illustrates the idea using a naive but valid solution to the decomposition technique on the LPE of Example 2.7.

Example 4.1. Reconsider the LPE *Machine* we defined earlier, and consider the two components depicted below.

$$\begin{aligned}
\mathrm{Machine}_V(n:Nat) = &\sum_{s:Bool} (n>0) \to \mathsf{count}|\mathsf{sync}^0_V(n,s)\,.\,\mathrm{Machine}_V(n-1)\\
&+ \sum_{s:Bool} (n\approx 0) \to \tau|\mathsf{sync}^1_V(n,s)\,.\,\mathrm{Machine}_V(3)\\
\mathrm{Machine}_W(s:Bool) = &\sum_{n:Nat} (n>0) \to \tau|\mathsf{sync}^0_W(n,s)\,.\,\mathrm{Machine}_W(s)\\
&+ \sum_{n:Nat} (n\approx 0) \to \mathsf{toggle}(s)|\mathsf{sync}^1_W(n,s)\,.\,\mathrm{Machine}_W(\neg s)
\end{aligned}$$

Each component describes part of the behaviour and knows the value of parameter n or s, but not the other. To cater for this, it is 'over-approximated' by a sum variable. The state space of $\mathrm{Machine}_V(0)$ is shown below. The synchronisation actions sync expose the non-deterministically chosen values of the unknown parameters.



Enforcing synchronisation of the sync actions, the context C can be chosen as follows to achieve a valid decomposition:

$$\nabla_{\{⟅\mathsf{toggle}⟆,⟅\mathsf{count}⟆\}}(\tau_{\{\mathsf{sync}^0,\mathsf{sync}^1\}}(\Gamma_{\{\mathsf{sync}^0_V|\mathsf{sync}^0_W\to\mathsf{sync}^0,\mathsf{sync}^1_V|\mathsf{sync}^1_W\to\mathsf{sync}^1\}}(\mathrm{Machine}_V(0) \parallel \mathrm{Machine}_W(\mathsf{false}))))$$

Unfortunately the state space of $\mathrm{Machine}_W(\mathsf{false})$ in the above example is infinitely branching and it has no finite state space that is strongly bisimilar to it, rendering the decomposition useless in practice. We will subsequently develop a more robust solution.

4.1 Separation Tuples

To obtain a useful decomposition it can be beneficial to reduce the number of parameters that occur in the synchronisation actions, because these then become a visible part of the transitions in the state spaces of the individual components. In the worst case, as illustrated by LPE Machine_W of Example 4.1, synchronisation actions lead to a component having an infinite state space despite the fact that the state space of the original LPE is finite.

One observation we exploit is that in some cases we can actually remove the synchronisation for summands completely. For instance, in the first summand of Machine in Example 2.7, the value of parameter s remains unchanged and the condition is only an expression containing parameter n. We refer to summands with such a property as *independent* summands. When defining the context C, we can allow a component to execute multi-actions of its independent summands without enforcing a synchronisation with the other component. This allows, for instance, component Machine_V to independently execute (multi-)action count without synchronising the values of s and n with Machine_W.

However, this might introduce an issue when (some of) the action labels of the independent summand also occur in action expressions of *other* summands. To see this, consider an LPE that contains two summands with a as action expression that are independent in different components and another summand with $a|a$ as action expression. In this case, we expect $a|a$ to be allowed in the composition expression to allow transitions from the last summand to occur. However, then the two independent summands in different components could also result in a simultaneous transition due to rule Par and that transition was not possible in the monolithic LPE. We solve the problem by introducing a *tag* action label and only allow the execution of independent actions with a single tag in the composition.

A second observation that we exploit is that if there are independent summands, then not every summand needs to be present in both components. However, we must ensure that each summand of the monolithic LPE is covered by at least one of the two components that we extract from the LPE. The summands that we extract for a given component are identified by a set of indices J of the summands of the monolithic LPE. Of these, we furthermore can identify summands that are dependent and summands that are independent. The indices for the latter are collected in the set K.

A third observation that can be utilised is that for the dependent summands, there is some degree of flexibility for deciding which part of the summand of the monolithic LPE will be contributed by which component. More specifically, by carefully distributing the enabling condition c and action expression α of a summand of the monolithic LPE over the two components, the amount of information (*i.e.*, information about 'missing' parameters, given by a synchronisation expression h) that needs to be exchanged between these two components when they execute their respective summands, can be minimised.

Note that the way we distribute the list of process parameters of the monolithic LPE over the two components may affect which summands can be considered independent. For instance, had we decided to assign the (multi-)action count to Machine_W and toggle to Machine_V, we would not be able to declare count's summand independent. Consequently, the set of process parameter indices U, assigned to a component, and the set K are mutually dependent. To capture this relation, we introduce the concept of a *separation tuple*. The concept of a separation tuple, a 7-tuple which we introduce below, formalises the required relation between K, J and U, and the conditions c, and action α and synchronisation expressions h of a component. To define the expressions we use indexed sets where the index of each element, indicated by a subscript, determines the index of the summand to which the expression belongs.

Definition 4.2. Let $P(\vec{d}:\vec{D}) = \mathop{+}_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE. Let $(P,U,K,J,c^U,\alpha^U,h^U)$ be a *separation* tuple such that $U \subseteq \mathbb{N}$ is a set of parameter indices and $K \subseteq J \subseteq I$ are two sets of summand indices. Furthermore, c^U, α^U and h^U are indexed sets of condition, action and update expressions

respectively such that for all $i \in (J \setminus K)$ it holds that $\mathsf{FV}(c_i^U) \cup \mathsf{FV}(\alpha_i^U) \cup \mathsf{FV}(\vec{h}_i^U) \subseteq \mathsf{Vars}(\vec{d}) \cup \{e_i\}$. Finally, for all $i \in K$ it holds that $\mathsf{FV}(c_i) \cup \mathsf{FV}(\alpha_i) \cup \mathsf{FV}(\vec{g}_{i|U}) \subseteq \mathsf{Vars}(\vec{d}_{|U}) \cup \{e_i\}$.

The separation tuple induces an LPE, where $U^c = \mathbb{N} \setminus U$, as follows:

$$P_U(\vec{d}_{|U} : \vec{D}_{|U}) = \bigoplus_{i \in (J \setminus K)} \sum_{e_i:E_i, \vec{d}_{|U^c}:\vec{D}_{|U^c}} c_i^U \to \alpha_i^U |\mathsf{sync}_U^i(\vec{h}_i^U) \,.\, P_U(\vec{g}_{i|U})$$
$$+ \bigoplus_{i \in K} \sum_{e_i:E_i} c_i \to \alpha_i |\mathsf{tag} \,.\, P_U(\vec{g}_{i|U})$$

We assume that action labels sync_V^i and sync_W^i, for any $i \in I$, and label tag does not occur in α_j, for any $j \in I$, to ensure that these action labels are fresh.

The components, induced by two separation tuples, may be combined in a context that enforces synchronisation of the sync events and hiding their communication trace so that all actions left can be traced back to the monolithic LPE from which the components originate. Under specific conditions, this is achieved by the following context.

Definition 4.3. Let $P(\vec{d} : \vec{D}) = \bigoplus_{i \in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and $(P, V, K_V, J_V, c^V, \alpha^V, h^V)$ and $(P, W, K_W, J_W, c^W, \alpha^W, h^W)$ be separation tuples. Let $P_V(\vec{d}_{|V} : \vec{D}_{|V}) = \phi_V$ and $P_W(\vec{d}_{|W} : \vec{D}_{|W}) = \phi_W$ be the induced LPEs according to Definition 4.2. Let $\iota : \vec{D}$ be a closed expression. Then the composition expression is defined as:

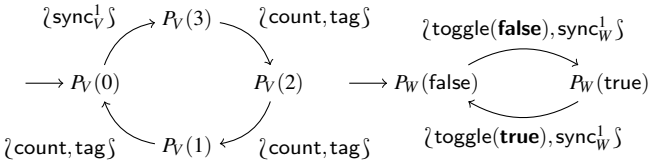
$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \mid i \in (K_V \cup K_W)\}}(\tau_{\{\mathsf{sync}^i \mid i \in I\}}(\Gamma_{\{\mathsf{sync}_V^i|\mathsf{sync}_W^i \to \mathsf{sync}^i \mid i \in I\}}(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})))))$$

Before we proceed to identify the conditions under which two separation tuples induce a valid decomposition using the above context, we revisit Example 2.7 to illustrate the concepts introduced so far.

Example 4.4. Reconsider the LPE presented in Example 2.7. The separation tuple $(P, V, \{0\}, \{0,1\}, \{(n > 0)_0, (n \approx 0)_1\}, \{(\mathsf{count}|\mathsf{tag})_0, (\tau)_1\}, \{\langle\rangle_1\})$ induces component P_V and $(P, W, \emptyset, \{1\}, \{\mathsf{true}_1\}, \{\mathsf{toggle}(s)_1\}, \{\langle\rangle_1\})$ component P_W.

$$P_V(n : Nat) = (n > 0) \to \mathsf{count}|\mathsf{tag} \,.\, P_V(n-1)$$
$$+ (n \approx 0) \to \mathsf{sync}_V^1 \,.\, P_V(3)$$
$$P_W(s : Bool) = \mathsf{true} \to \mathsf{toggle}(s)|\mathsf{sync}_W^1 \,.\, P_W(\neg s)$$

Note that we omitted the $\sum$-operator in the summands of P_V since sum variable s does not occur as a free variable in the expressions; and similar reasoning for P_W. The state spaces of components $P_V(0)$ and $P_W(\mathsf{false})$ are shown below.



We obtain the following composition according to Definition 4.3:

$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\langle\mathsf{toggle}\rangle, \langle\mathsf{count}\rangle, \langle\mathsf{count,tag}\rangle\}}(\tau_{\{\mathsf{sync}^0, \mathsf{sync}^1\}}($$
$$\Gamma_{\{\mathsf{sync}_V^0|\mathsf{sync}_W^0 \to \mathsf{sync}^0, \mathsf{sync}_V^1|\mathsf{sync}_W^1 \to \mathsf{sync}^1\}}(P_V(0) \parallel P_W(\mathsf{false})))))$$

The state space of this expression is strongly bisimilar to the state space of Machine$(0, \mathsf{false})$ shown in Example 2.7. As shown above the state space of $P_V(0)$ has four states and transitions, and the state space of $P_W(\mathsf{false})$ has two states and transitions, which are both smaller than the original state space. Their composition has the same size as the original state space and no further minimisation can be achieved.

4.2 Cleave Correctness Criteria

It may be clear that not every decomposition which satisfies Definition 4.3 yields a *valid* decomposition (in the sense of Definition 3.1). For example, replacing the condition expression true in Example 4.4 of the summand in P_W by false would not result in a valid decomposition. Our aim in this section is to present the necessary and sufficient conditions to establish that the state space of the monolithic LPE is bisimilar to the state space of the composition expression resulting from Definition 4.3.

Consider any decomposition of an LPE P according to Definition 4.3, induced by separation tuples $(P,V,K_V,J_V,c^V,\alpha^V,h^V)$ and $(P,W,K_W,J_W,c^W,\alpha^W,h^W)$. Note that bisimilarity requires states that are related to mimick each other's steps. Since three LPEs are involved (the LPE P and the two components induced by the separation tuples), we must consider situations that can emerge from any of these three LPEs executing a (multi-)action.

Suppose that the monolithic LPE P can take a step; this must be because the enabling condition of a summand of P holds true. Then that same summand must appear in at least one component induced by a separation tuple. Vice versa, in case one of the components can take a step, then we must be able to trace back that step to the monolithic LPE. In case the summand is dependent, then we must ensure that that summand is covered by both components, and, together, both components cover all summands. This gives rise to requirement (SYN): $J_V = I \setminus K_V$ and $J_W = I \setminus K_W$.

For the independent summands with indices K_V, we see that Definition 4.3 guarantees that the free variables of their condition, action and update expressions are taken from $\vec{d}_{|V}$. Dually for the independent summands related to K_W. However, this is not sufficient to guarantee full independence of both components: what may happen is that the execution of a summand that is assumed to be independent still modifies the value of a process parameter of the other component, violating the idea of independence. In order to guarantee true independence, we add requirement (IND): for all $r \in K_V$ we have $\vec{g}_{r|W} = \vec{d}_{|W}$, and, dually, for all $r \in K_W$ we demand $\vec{g}_{r|V} = \vec{d}_{|V}$. Note that in case K_V and K_W overlap, condition (IND) guarantees that the involved summands only induce self-loops.

For the dependent summands, *i.e.*, those with an index r in $(J_V \cap J_W)$, both components must necessarily execute their r-indexed summands to match the r-indexed summand of P. Note that the enabledness of summand r in P depends on the enabling condition c_r. Consequently, if c_r holds true, then the r-indexed conditions c_r^V and c_r^W must also hold true. Moreover, since we are dealing with dependent summands, the multi-action expression $\alpha_r^V | \alpha_r^W$ must reduce to α_r under these conditions. It is important to observe that also the additional synchronisation vectors $\vec{h}^V$ and $\vec{h}^W$ must agree, for otherwise the sync actions of both components cannot participate in the synchronisation. Note that we do not need to explicitly require that the next state reached by P after executing its r-indexed summand can also be reached by the individual components upon executing their r-indexed summands, since this property is guaranteed by the construction in Definition 4.2. These requirements are guaranteed by condition (ORI). Vice versa, whenever both components can simultaneously execute their r-indexed summand, we must ensure that also the monolithic LPE P can execute its r-indexed summand. Condition (COM) ensures that this requirement is met. A technical complication in formalising requirement (COM), however, is that the sum variables of the individual components carry the same name in all three LPEs. In particular, from the fact that both individual components can successfully synchronise, we cannot deduce a unique value assigned to these homonymous sum-variables. We must therefore also ensure that the resulting next states reached in the components by executing the r-indexed summands indeed is the same as could have been reached by executing the r-indexed summand in P. Contrary to requirement (ORI), this property is not guaranteed by the construction of Definition 4.2, so there is a need to explicitly require it to hold.

A pair of separation tuples of P satisfying the above requirements is called a *cleave* of P. Below, we

formalise this notion, together with the requirements we informally introduced above.

Definition 4.5. Let $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and $(P,V,K_V,J_V,c^V,\alpha^V,h^V)$ and $(P,W,K_W,J_W,c^W,\alpha^W,h^W)$ be separation tuples as defined in Definition 4.2. The two separation tuples are a *cleave* of P iff the following requirements hold.

SYN. $J_V = I \setminus K_W$ and $J_W = I \setminus K_V$.

IND. For all $r \in K_V$, $\vec{g}_{r|W} = \vec{d}_{|W}$, and for all $r \in K_W$, $\vec{g}_{r|V} = \vec{d}_{|V}$.

ORI. For all $r \in (J_V \cap J_W)$ and substitutions σ satisfying $[\![\sigma(c_r)]\!]$, also:

- $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma(c_r^W)]\!]$, and
- $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma(\vec{h}_r^W))]\!]$, and
- $[\![\sigma(\alpha_r^V|\alpha_r^W)]\!] = [\![\sigma(\alpha_r)]\!]$.

COM. For all $r \in (J_V \cap J_W)$ and substitutions σ and σ' satisfying $[\![\sigma(c_r^V)]\!]$ and $[\![\sigma'(c_r^W)]\!]$ and $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$, there is a substitution ρ such that $[\![\rho(\vec{d}_{|V})]\!] = [\![\sigma(\vec{d}_{|V})]\!]$ and $[\![\rho(\vec{d}_{|W})]\!] = [\![\sigma'(\vec{d}_{|W})]\!]$ and:

- $[\![\rho(c_r)]\!]$, and
- $[\![\sigma(\alpha_r^V)|\sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r)]\!]$, and
- $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$, and
- $[\![\sigma'(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$.

Observe that the separation tuples inducing the decomposition obtained Example 4.4 are a cleave indeed. Moreover, we already argued informally that the decomposition yields a state space that is bisimilar to the original monolithic LPE. We remark that also the decomposition we obtained in Example 4.1 can be achieved by a cleave. However, we can also observe that this decomposition is not a particularly useful cleave for the purpose of compositional minimisation.

We finish this section with a formal claim stating that a cleave induces a valid decomposition of a monolithic LPE. The complete proof can be found in Appendix A.

Theorem 4.6. Let $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(P,V,K_V,J_V,c^V,\alpha^V,h^V)$ and $(P,W,K_W,J_W,c^W,\alpha^W,h^W)$ be a cleave as defined in Definition 4.5. The composition expression defined in Definition 4.3 is a valid decomposition as defined in Definition 3.1 for a closed expression $\vec{\iota}'' : \vec{D}$.

Proof. We show that $[\![P(\vec{\iota}'')]\!]$ is strongly bisimilar to $[\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}''_{|V}) \parallel P_W(\vec{\iota}''_{|W})))))]\!]$ (the composition expression of Definition 4.3) where $H' = \{\mathsf{tag}\}$, $A = \{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \mid i \in (K_V \cup K_W)\}$, $H = \{\mathsf{sync}^i \mid i \in I\}$ and $C = \{\mathsf{sync}^i_V|\mathsf{sync}^i_W \to \mathsf{sync}^i \mid i \in I\}$ and therefore a valid decomposition.

Let $(S_1,s_1,Act_1,\to_1) = [\![P(\vec{\iota}'')]\!]$ and $(S_2,s_2,Act_2,\to_2) = [\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}''_{|V}) \parallel P_W(\vec{\iota}''_{|W})))))]\!]$. Let R be the smallest relation such that $P(\vec{\iota}')\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}'_{|V}) \parallel P_W(\vec{\iota}'_{|W})))))$ for any closed expression $\vec{\iota}' : \vec{D}$. We show that R is a strong bisimulation relation up to $\leftrightarrows$. Pick any arbitrary closed expression $\vec{\iota} : \vec{D}$ and suppose that $P(\vec{\iota})\ R\ \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})))))$ holds.

- Case $P(\vec{\iota}) \xrightarrow{\omega}_1 Q'$. There is an index $r \in I$ and a closed expression $l : E_r$ such that for $\sigma = [\vec{d} \leftarrow \vec{\iota}, e_r \leftarrow l]$ it holds that $[\![\sigma(c_r)]\!]$, $\omega = [\![\sigma(\alpha_r)]\!]$ and $Q' = P(\sigma(\vec{g}_r))$. There are three cases to consider based on the index r.
  - Case $r \in I \setminus (K_V \cup K_W)$. From SYN this means that $r \in (J_V \cap J_W)$. We derive the transitions using requirement ORI. First, $[\![\sigma(c_r^V)]\!]$ and $[\![c_r^W)]\!]$ and therefore:

$$P_V(\vec{\iota}_{|V}) \xrightarrow{[\![\sigma(\alpha_r^V|\mathsf{sync}_r^V(\vec{h}_r^V))]\!]}_2 P_V(\sigma(\vec{g}_{r|V})) \text{ and } P_W(\vec{\iota}_{|W}) \xrightarrow{[\![\sigma(\alpha_r^W|\mathsf{sync}_r^W(\vec{h}_r^W))]\!]}_2 P_W(\sigma(\vec{g}_{r|W}))$$

Furthermore, $[\![\sigma(\alpha_r)]\!] = [\![\sigma(\alpha_r^V|\alpha_r^W)]\!]$ and by rule PAR there is:

$$P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{sync}_V(\sigma(\vec{h}_r^V))|\mathsf{sync}_W(\sigma(\vec{h}_r^W))]\!]}_2 P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))$$

From $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma(\vec{h}_r^W)]\!]$ follows $\theta_{H'}(\theta_H(\gamma_C([\![\sigma(\alpha_r)|\mathsf{sync}_V(\sigma(\vec{h}_r^V))|\mathsf{sync}_W(\sigma(\vec{h}_r^W))]\!]))) = [\![\sigma(\alpha_r)]\!]$ and from $\underline{[\![\sigma(\alpha_r)]\!]} \in A$ we can derive using the operational rules that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}))))) \xrightarrow{[\![\sigma(\alpha_r)]\!]}_2 \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W})))))))$$

Finally, $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.

- Case $r \in K_V$. In short, we derive $P_V(\vec{\imath}_{|V}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{tag}]\!]}_2 P_V(\sigma(\vec{g}_{r|V}))$ because $[\![\sigma(c_r)]\!]$ holds. The composition expression has a transition labelled $[\![\sigma(\alpha_r)]\!]$ to the state $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$ from PARL, IND and the definition of auxiliary operators. Finally, by definition $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.
- Case $r \in K_W$. Similar to case $r \in K_V$.

- Case $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}))))) \xrightarrow{\omega}_2 Q'$. We can derive that there is an expression $Q \in \mathsf{S}$ such that $Q' = \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$, and $\omega' \in \Omega$ such that $\omega = \theta_{H'}(\theta_H(\gamma_C(\omega')))$, $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 Q$ and $\underline{\omega} \in A$. There are three cases where a parallel composition results in a transition. Suppose that $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 Q$ is due to:
  - Rule PAR with premise $P_V(\vec{\imath}_{|V}) \xrightarrow{\omega_V}_2 P_V'$ and $P_W(\vec{\imath}_{|W}) \xrightarrow{\omega_W}_2 P_W'$. Then there is a transition $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega_V+\omega_W}_2 P_V' \parallel P_W'$ such that $\omega_V + \omega_W = \omega'$ and $Q = P_V' \parallel P_W'$. From $\underline{\theta_{H'}(\theta_H(\gamma_C(\omega_V + \omega_W)))} \in A$ we know that $\gamma_C(\omega_V + \omega_W) = \omega + \wr\mathsf{sync}_r\wr$, for some index $r \in I$, because only a single tag is allowed with original action labels. Therefore, it also follows that $r \in I \setminus (K_V \cup K_W)$.
    Observe that the sets of indices V and V^c are disjoint (and similarly W and W^c). Also $\vec{d}_{|V}$ are the parameters of P_V and $\vec{d}_{|V^c}$ are sum variables in the r-indexed summand for which any value can be chosen. Therefore, by the existence of these transitions in the premise there are closed expressions $l, l' : E_r$, and $\vec{\kappa} : \vec{D}_{|V^c}$, $\vec{\kappa}' : \vec{D}_{|W^c}$ with the substitutions $\sigma = [\vec{d}_{|V} \leftarrow \vec{\imath}_{|V}, \vec{d}_{|V^c} \leftarrow \vec{\kappa}, e_r \leftarrow l]$ and $\sigma' = [\vec{d}_{|W} \leftarrow \vec{\imath}_{|W}, \vec{d}_{|W^c} \leftarrow \vec{\kappa}', e_r \leftarrow l']$ such that $[\![\sigma(c_r^V)]\!]$ holds, $\omega_V = [\![\sigma(\alpha_r^V|\mathsf{sync}_r^V(\vec{h}_r^V))]\!]$ and $P_V' = P_V([\![\sigma(\vec{g}_{r|V})]\!])$. Furthermore, $[\![\sigma'(c_r^W)]\!]$ holds, $\omega_W = [\![\sigma'(\alpha_r^W|\mathsf{sync}_r^W(\vec{h}_r^W))]\!]$ and $P_W' = P_W([\![\sigma'(\vec{g}_{r|W})]\!])$. Finally, $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$ due to synchronisation.
    From the requirement COM it follows that there is a closed expression $l'' : E_r$ such that for $\rho = [\vec{d} \leftarrow \vec{\imath}, e_r \leftarrow l'']$ that $[\![\rho(c_r)]\!]$ holds and $[\![\sigma(\alpha_r^V)|\sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r)]\!]$. We conclude that $P(\vec{\imath}) \xrightarrow{\omega}_1 P(\rho(\vec{g}_r))$. Furthermore, from $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$, and $[\![\sigma'(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$ and Lemma 2.6 it follows that $P_V(\sigma(\vec{g}_{r|V})) \leftrightarrows P_V(\rho(\vec{g}_{r|V}))$ and $P_W(\sigma'(\vec{g}_{r|W})) \leftrightarrows P_W(\rho(\vec{g}_{r|W}))$ By the congruence of strong bisimilarity with respect to S we obtain that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma'(\vec{g}_{r|W})))))) \leftrightarrows \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W}))))))$$

Finally, $P(\rho(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W}))))))$.

– Rule PARL with premise $P_V(\vec{\imath}_{|V}) \xrightarrow{\omega'}_2 P'_V$. Then $Q = P'_V \parallel P_W(\vec{\imath}_{|W})$. Pick an arbitrary index $r \in J_V$. If $r \in J_V \setminus K_V$ then the action expression contains an action labelled sync_r^V, which means that $\underline{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \notin A$. Contradiction, and thus $r \in K_V$. Now, we can show that $P' = P_V(\sigma(\vec{g}_{r|V}))$ and that there is a transition $P(\vec{\imath}) \xrightarrow{\omega'}_1 P(\sigma(\vec{g}_r))$ by IND for a suitable substitution σ. Finally, $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.

– Rule PARR with premise $P_V(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 P'_W$, along the same lines as above.

We conclude that $[\![P(\vec{\imath}''_{|V})]\!] \leftrightarroot \, [\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\imath}''_{|V}) \parallel P_W(\vec{\imath}''_{|W})))))]\!]$. □

5 State Invariants

One way to restrict the behaviour of the components is to strengthen the condition expressions of each summand to avoid certain outgoing transitions. We show that so-called *state invariants* [1] can be used for this purpose. These state invariants are typically formulated by the user based on intuition of the model behaviour.

Definition 5.1. Given an LPE $P(d:D) = \sum_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(g_i)$. A boolean expression ψ such that $\mathsf{FV}(\psi) \subseteq \{d\}$ is called a *state invariant* iff the following holds: for all $i \in I$ and closed expressions $\iota : D$ and $l : E_i$ such that $[\![[d \leftarrow \iota, e_i \leftarrow l](c_i \wedge \psi)]\!]$ holds then $[\![[d \leftarrow [d \leftarrow \iota, e_i \leftarrow l](g_i)](\psi)]\!]$ holds as well.

The essential property of a state invariant is that whenever it holds for the initial state it is guaranteed to hold for all reachable states in the state space. This follows relatively straightforward from its definition. Next, we define a *restricted* LPE where (some of) the condition expressions are strengthened with a boolean expression.

Definition 5.2. Given an LPE $P(d:D) = \sum_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(g_i)$, a boolean expression ψ such that $\mathsf{FV}(\psi) \subseteq \{d\}$ and a set of indices $J \subseteq I$. We define the restricted LPE, denoted by $P^{\psi,J}$, as follows:

$$P^{\psi,J}(d:D) = \sum_{i\in J} \sum_{e_i:E_i} c_i \wedge \psi \to \alpha_i \,.\, P^{\psi,J}(g_i) + \sum_{i\in (I\setminus J)} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P^{\psi,J}(g_i)$$

Note that if the boolean expression ψ in Definition 5.2 is a state invariant for the given LPE then for all closed expressions $\vec{\imath} : \vec{D}$ such that $[\![[\vec{d} \leftarrow \vec{\imath}](\psi)]\!]$ holds, it holds that $[\![P(\vec{\imath})]\!] \leftrightarroot \, [\![P^{\psi,J}(\vec{\imath})]\!]$, for any $J \subseteq I$. Therefore, we can use a state invariant of an LPE to strengthen all of its condition expressions.

Moreover, a state invariant of the original LPE can *also* be used to restrict the behaviour of the components obtained from a cleave, as formalised in the following theorem. Note that the set of indices is used to only strengthen the condition expressions of summands that introduce synchronisation, because the condition expressions of independent summands cannot contain the other parameters as free variables. Furthermore, the restriction can be applied to independent summands before the decomposition.

Theorem 5.3. Let $P(\vec{d} : \vec{D}) = \sum_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and $(V, K_V, J_V, c^V, \alpha^V, h^V)$ and $(W, K_W, J_W, c^W, \alpha^W, h^W)$ be separations tuples as defined in Definition 4.2. Let ψ be a state invariant of P. Given closed expressions $\vec{\imath} : \vec{D}$ such that $[\![[\vec{d} \leftarrow \vec{\imath}](\psi)]\!]$ holds the following expression, where $C = J_V \cap J_W$, is a valid decomposition:

$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \mid i\in I\}\cup\{\underline{\alpha_i|\mathsf{tag}} \mid i\in (Ind_V \cup K_W)\}}(\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\Gamma_{\{\mathsf{sync}^i_V \mid \mathsf{sync}^i_W \to \mathsf{sync}^i \mid i\in I\}}(P_V^{\psi,C}(\vec{\imath}_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}_{|W})))))$$

Proof. This proof is similar to the proof of Theorem 4.6 with the strong bisimilation relation only defined for states where the invariant holds. The full proof can be found in Appendix B. □

Observe that the predicate $n \leq 3$ is a state invariant of the LPE Machine in Example 2.7. Therefore, we can consider the process $\text{Machine}_W^{\psi,I}$ in Example 4.1 for the composition expression, which is finite. This would yield two finite components, but the state space of $\text{Machine}_W^{\psi,I}$ is larger than that of P_W in Example 4.4.

Finally, we remark that the restricted state space contains deadlock states whenever the invariant does not hold. These deadlocks can be avoided by applying the invariant to the update expression of each parameter instead of the parameter itself without affecting the correctness.

6 Implementation

While Theorem 4.6 and Definition 4.5 together provide the conditions that guarantee that a cleave yields a valid decomposition, requirements (ORI) and (COM) of the latter definition are difficult to ensure due to the semantic nature of these requirements. In practice, we need to effectively approximate these correctness requirements using static analysis.

While we leave it to future work to investigate to what extent a precise and efficient static analysis is possible, we have implemented an automated prototype translation that, given a user-supplied partitioning of the process parameters of the monolithic LPE, exploits a simple static analysis to obtain components that are guaranteed to satisfy the requirements of a cleave. First of all the prototype identifies the independent summands in both components. Furthermore, we decide on each clause of a conjunctive condition and action in the action expression where it belongs. This analysis is based on the observation that whenever all free variables of an expression occur in one component then that expression should be kept in that component, and thus removed from the other component.

7 Case Study

We have used our prototype to carry out several experiments using specifications written in the high-level language mCRL2 [11], a process algebra generalising the one of Section 2.3. To apply the decomposition technique we use the LPEs that the mCRL2 toolset [4] generates as part of the pre-processing step the toolset uses before further analyses of the specifications are conducted. We compare the results of the monolithic exploration and the exploration based on the decomposition technique.

7.1 Alternating Bit Protocol

The alternating bit protocol (ABP) is a communication protocol that uses a single control bit, which is sent along the message, to implement a reliable communication channel over two unreliable channels [11]. The specification contains four processes for the sender, receiver and two unreliable communication channels.

First, we choose the partitioning of the parameters such that one component (ABP_V) contains the parameters of the sender and one communication channel, and the other component (ABP_W) contains the parameters of the receiver and the other communication channel. We observe that both components are larger than the original state space, and can also not be minimised further, illustrating that traditional compositional minimisation is, in this case, not particularly useful.

Table 1: Metrics for the alternating bit protocol.

Model	original		minimised	
	#states	#trans	#states	#trans
ABP	182	230	48	58
ABP_V	204	512	204	512
ABP_W	64	196	60	192
ABP_V^{ψ}	104	180	53	180
ABP_W^{ψ}	58	110	21	108
$\mathsf{ABP}_V^{\psi} \parallel \mathsf{ABP}_W^{\psi}$	172	220	48	58
ABP'_V	5	35	5	35
ABP'_W	78	118	28	42
$\mathsf{ABP}'_V \parallel \mathsf{ABP}'_W$	76	90	48	58

Further analysis showed that the behaviour of each process heavily depends on the state of the other processes, which results in large components as this information is lost. We can encode this global information as a state invariant based on the *control flow* parameters. The second cleave is obtained by obtaining two restricted components (ABP_V^{ψ} and ABP_W^{ψ}) using this invariant. This yields a useful decomposition. Finally, we have obtained a cleave into components ABP'_V and ABP'_W where the partitioning is not based on the original processes. This yields a very effective cleave as shown in Table 1.

7.2 Practical Examples

Table 2: State space metrics for various practical specifications.

Model	Ref	exploration		minimised	
		#states	#transitions	#states	#transitions
Chatbox	[18]	65 536	2 621 440	16	144
$\mathsf{Chatbox}_V$		128	4 352	128	3 456
$\mathsf{Chatbox}_W$		512	37 888	8	440
$\mathsf{Chatbox}_V \parallel \mathsf{Chatbox}_W$		1 024	22 528	16	144
Register	[12]	914 048	1 885 824	1 740	3 572
$\mathsf{Register}_V$		464	10 672	464	10 672
$\mathsf{Register}_W$		97 280	273 408	5 760	16 832
$\mathsf{Register}_V \parallel \mathsf{Register}_W$		76 416	157 952	1 740	3 572
WMS	[17]	155 034 776	2 492 918 760	44 526 316	698 524 456
WMS_V		212 992	5 144 576	212 992	2 801 664
WMS_W		1 903 715	121 945 196	414 540	26 429 911
$\mathsf{WMS}_V \parallel \mathsf{WMS}_W$		64 635 040	1 031 080 812	44 526 316	698 524 456

The Chatbox specification [18] describes a chat room facility in which four users can join, leave and send messages. This specification is interesting because it is described as a monolithic process, which

means that compositional minimisation is not applicable in the first place. The size of the components ($\mathsf{Chatbox}_V$ and $\mathsf{Chatbox}_W$) before and after minimisation modulo strong bisimulation are presented to show that these are small and can be further reduced. The composition $(\mathsf{Chatbox}_V \parallel \mathsf{Chatbox}_W)$ shows that indeed the decomposition technique can be used quite successfully, because the result under exploration is much smaller than the original state space. Finally, we have also listed the size of the minimised original state space (which is equal to the minimised composition) as it indicates the best possible result. The Register specification [12] describes a wait-free handshake register and the WMS specification a workload management system [17], used at CERN. For the latter two experiments we found that partitioning the parameters into a set of so-called *control flow parameters* and remaining parameters yields the best results.

We also consider the total execution time and maximum amount of memory required to obtain the original state space using exploration and the state space obtained using the decomposition technique, for which the results can be found in Table 3. The execution times in **s**econds or **h**ours required to obtain the state space under "exploration" in Table 2, excluding the final minimisation step of the original or composition state space which are only shown for reference. The cost of the static analysis of the cleave itself was in the range of several milliseconds.

Table 3: Execution times and maximum memory usage measurements.

Model	monolithic		decomposition	
	time	memory	time	memory
Chatbox	4.76s	21.9MB	0.2s	15.7MB
Register	7.94s	99.7MB	1.56s	47.7MB
WMS	2.4h	15.1GB	0.8h	11.8GB

8 Conclusion

We have presented a decomposition technique, referred to as cleave, that can be applied to any monolithic process with the structure of an LPE and have shown that the result is always a valid decomposition. Furthermore, we have shown that state invariants can be used to improve the effectiveness of the decomposition. We consider defining a static analysis to automatically derive the parameter partitioning for the practical application of this technique as future work. Furthermore, the cleave is currently not well-suited for applying the typically more useful abstraction based on (divergence-preserving) branching bisimulation minimisation [8]. The reason for this is that τ-actions might be extended with synchronisation actions and tags. As a result they become visible, effectively reducing branching bisimilarity to strong bisimilarity.

References

[1] Marc Bezem & Jan Friso Groote (1994): *Invariants in Process Algebra with Data*. In Bengt Jonsson & Joachim Parrow, editors: *CONCUR*, *LNCS* 836, Springer, pp. 401–416, doi:10.1007/978-3-540-48654-1_30. Available at `https://doi.org/10.1007/978-3-540-48654-1_30`.

[2] Pierre Bouvier, Hubert Garavel & Hernán Ponce de León (2020): *Automatic Decomposition of Petri Nets into Automata Networks - A Synthetic Account*. In Ryszard Janicki, Natalia Sidorova & Thomas Chatain, editors: *Application and Theory of Petri Nets and Concurrency - 41st International Conference, PETRI NETS 2020,*

Paris, France, June 24-25, 2020, Proceedings, *Lecture Notes in Computer Science* 12152, Springer, pp. 3–23, doi:10.1007/978-3-030-51831-8_1. Available at https://doi.org/10.1007/978-3-030-51831-8_1.

[3] Ed Brinksma, Rom Langerak & Peter Broekroelofs (1993): *Functionality Decomposition by Compositional Correctness Preserving Transformation*. In Costas Courcoubetis, editor: *CAV*, *LNCS* 697, Springer, pp. 371–384, doi:10.1007/3-540-56922-7_31.

[4] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability*. In Tomás Vojnar & Lijun Zhang, editors: *TACAS*, *LNCS* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[5] Shing-Chi Cheung & Jeff Kramer (1996): *Context Constraints for Compositional Reachability Analysis*. *ACM Trans. Softw. Eng. Methodol.* 5(4), pp. 334–377, doi:10.1145/235321.235323.

[6] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe (2013): *CADP 2011: a toolbox for the construction and analysis of distributed processes*. *STTT* 15(2), pp. 89–107, doi:10.1007/s10009-012-0244-z.

[7] Hubert Garavel, Frédéric Lang & Laurent Mounier (2018): *Compositional Verification in Action*. In Falk Howar & Jiri Barnat, editors: *FMICS*, *LNCS* 11119, Springer, pp. 189–210, doi:10.1007/978-3-030-00244-2_13.

[8] R. J. van Glabbeek, B. Luttik & N. Trčka (2009): *Branching Bisimilarity with Explicit Divergence*. *Fundam. Inform.* 93(4), pp. 371–392, doi:10.3233/FI-2009-109.

[9] Susanne Graf, Bernhard Steffen & Gerald Lüttgen (1996): *Compositional Minimisation of Finite State Systems Using Interface Specifications*. *Formal Asp. Comput.* 8(5), pp. 607–616, doi:10.1007/BF01211911.

[10] Jan Friso Groote & Faron Moller (1992): *Verification of Parallel Systems via Decomposition*. In Rance Cleaveland, editor: *CONCUR*, *LNCS* 630, Springer, pp. 62–76, doi:10.1007/BFb0084783.

[11] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and Analysis of Communicating Systems*. MIT Press. Available at https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems.

[12] Wim H. Hesselink (1998): *Invariants for the Construction of a Handshake Register*. *Inf. Process. Lett.* 68(4), pp. 173–177, doi:10.1016/S0020-0190(98)00158-6. Available at https://doi.org/10.1016/S0020-0190(98)00158-6.

[13] Sung-Shik T. Q. Jongmans, Dave Clarke & José Proença (2016): *A procedure for splitting data-aware processes and its application to coordination*. *Sci. Comput. Program.* 115-116, pp. 47–78, doi:10.1016/j.scico.2014.02.017.

[14] Robin Milner (1983): *Calculi for Synchrony and Asynchrony*. *Theor. Comput. Sci.* 25, pp. 267–310, doi:10.1016/0304-3975(83)90114-7.

[15] Robin Milner (1989): *Communication and concurrency*. PHI Series in computer science, Prentice Hall.

[16] Jaco van de Pol & Mark Timmer (2009): *State Space Reduction of Linear Processes Using Control Flow Reconstruction*. In Zhiming Liu & Anders P. Ravn, editors: *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*, *Lecture Notes in Computer Science* 5799, Springer, pp. 54–68, doi:10.1007/978-3-642-04761-9_5. Available at https://doi.org/10.1007/978-3-642-04761-9_5.

[17] Daniela Remenska, Tim A. C. Willemse, Kees Verstoep, Wan J. Fokkink, Jeff Templon & Henri E. Bal (2012): *Using Model Checking to Analyze the System Behavior of the LHC Production Grid*. In: *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012, Ottawa, Canada, May 13-16, 2012*, IEEE Computer Society, pp. 335–343, doi:10.1109/CCGrid.2012.90. Available at https://doi.org/10.1109/CCGrid.2012.90.

[18] Judi Romijn & Jan Springintveld (1998): *Exploiting Symmetry in Protocol Testing*. In Stanislaw Budkowski, Ana R. Cavalli & Elie Najm, editors: *FORTE XI / PSTV XVIII*, *IFIP Conference Proceedings* 135, Kluwer, pp. 337–352.

[19] Kuo-Chung Tai & Pramod V. Koppol (1993): *Hierarchy-based incremental analysis of communication protocols*. In: *ICNP*, IEEE Computer Society, pp. 318–325, doi:10.1109/ICNP.1993.340896.

[20] Kuo-Chung Tai & Pramod V. Koppol (1993): *An Incremental Approach to Reachability Analysis of Distributed Programs*. In Jack C. Wileden, editor: *IWSSD*, IEEE Computer Society, pp. 141–151.

The appendix is only provided for completeness. These proofs will be made available in a preprint.

A Proof of Theorem 4.6

The following definition and proposition is due to [15]. These are in the context of two LTSs $\mathscr{L}_1 = (S_1, s_1, Act_1, \rightarrow_1)$ and $\mathscr{L}_2 = (S_2, s_2, Act_2, \rightarrow_2)$. We introduce for a binary relation $R \subseteq S_1 \times S_2$ the following notation $\leftrightarrows R \leftrightarrows$ to denote the *relational composition* such that $\leftrightarrows R \leftrightarrows = \{(s,t) \in S_1 \times S_2 \mid \exists s' \in S_1, t' \in S_2 : s \leftrightarrows s' \wedge s' \, R \, t' \wedge t' \leftrightarrows t\}$.

Definition A.1. A binary relation $R \subseteq S_1 \times S_2$ is a *strong bisimulation up to* $\leftrightarrows$ iff for all $s \, R \, t$ it holds that:

- if $s \xrightarrow{\omega} s'$ then there is a state $t' \in S_2$ such that $t \xrightarrow{\omega} t'$ and $s' \leftrightarrows R \leftrightarrows t'$.
- if $t \xrightarrow{\omega} t'$ then there is a state $s' \in S_1$ such that $t \xrightarrow{\omega} t'$ and $t' \leftrightarrows R \leftrightarrows s'$.

Proposition A.2. If R is a strong bisimulation up to $\leftrightarrows$ then $R \subseteq \leftrightarrows$

This result establishes that if R is a strong bisimulation up to $\leftrightarrows$ then for any pair $(s,t) \in R$ we can conclude that $s \leftrightarrows t$.

We introduce two auxiliary lemmas to relate the transition induced by some expression $P \in \mathsf{S}$ to the transitions induced by applying the allow, hide and communication operators, in the same order as the composition expression defined in Definition 4.3, to P.

Lemma A.3. Given expressions $P, Q \in \mathsf{S}$, a set of multi-sets of action labels $A \subseteq 2^{\Lambda \rightarrow \mathbb{N}}$, sets of events $H', H \subseteq \Lambda$, a set of communications $C \subseteq \mathsf{Comm}$. If $P \xrightarrow{\omega'} Q$ and $\underline{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \in A$ then:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$$

Proof. We can derive the following:

$$\textsc{Hide}\ \dfrac{\textsc{Allow}\ \dfrac{\textsc{Hide}\ \dfrac{\textsc{Com}\ \dfrac{P \xrightarrow{\omega'} Q \qquad C \subseteq \mathsf{Comm}}{\Gamma_C(P) \xrightarrow{\gamma_C(\omega')} \Gamma_C(Q)} \qquad H \subseteq \Lambda}{\tau_H(\Gamma_C(P)) \xrightarrow{\theta_H(\gamma_C(\omega'))} \tau_H(\Gamma_C(Q))} \qquad A \subseteq 2^{\Lambda \rightarrow \mathbb{N}} \qquad \underline{\theta_H(\gamma_C(\omega'))} \in A}{\nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\theta_H(\gamma_C(\omega'))} \nabla_A(\tau_H(\Gamma_C(Q)))} \qquad H' \subseteq \Lambda}{\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))}$$

□

Lemma A.4. Given expressions $P, Q \in \mathsf{S}$, a set of multi-sets of action labels $A \subseteq 2^{\Lambda \rightarrow \mathbb{N}}$, sets of events $H', H \subseteq \Lambda$, a set of communications $C \subseteq \mathsf{Comm}$ if:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q'$$

then there are $Q \in \mathsf{S}$ and $\omega' \in \Omega$ such that $Q' = \nabla_A(\tau_H(\Gamma_C(Q)))$, $\omega = \theta_{H'}(\theta_H(\gamma_C(\omega')))$, $P \xrightarrow{\omega'} Q$ and $\underline{\omega} \in A$.

Proof. Assume that $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\omega} Q'$. From the structure of the premise conclusion we know that only rule HIDE is applicable, which can only be applied for some $Q'' \in \mathsf{S}$ such that:

$$\textsc{Hide}\ \frac{\nabla_A(\tau_H(\Gamma_C(P))) \xrightarrow{\theta_{H'}(\omega)} Q'' \qquad H' \subseteq \Lambda}{\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P)))) \xrightarrow{\theta_{H'}(\omega)} \tau_{H'}(Q'')} \tag{1}$$

Similarly, we derive the applicability of the ALLOW and COM rules such that we essentially can obtain (the only possible) derivation shown in the proof of Lemma A.3. □

Theorem 4.6. Let $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(P,V,K_V,J_V,c^V,\alpha^V,h^V)$ and $(P,W,K_W,J_W,c^W,\alpha^W,h^W)$ be a cleave as defined in Definition 4.5. The composition expression defined in Definition 4.3 is a valid decomposition as defined in Definition 3.1 for a closed expression $\vec{\iota}'' : \vec{D}$.

Proof. Let $P(\vec{d}:\vec{D}) = +_{i\in I}\sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and let $(P,V,K_V,J_V,c^V,\alpha^V,h^V)$ and $(P,W,K_W,J_W,c^W,\alpha^W,h^W)$ be a cleave as defined in Definition 4.5.

Pick an arbitrary closed expression $\vec{\iota}'' : \vec{D}$. We show that $[\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}''_{|V}) \parallel P_W(\vec{\iota}''_{|W})))))]\!]$ where $H' = \{\mathsf{tag}\}$, $A = \{\underline{\alpha_i} \mid i \in I\} \cup \{\underline{\alpha_i|\mathsf{tag}} \mid i \in (K_V \cup K_W)\}$, $H = \{\mathsf{sync}^i \mid i \in I\}$ and $C = \{\mathsf{sync}^i_V|\mathsf{sync}^i_W \to \mathsf{sync}^i \mid i \in I\}$ is strongly bisimilar to $[\![P(\vec{\iota}'')]\!]$.

Let $(S_1,s_1,Act_1,\to_1) = [\![P(\vec{\iota}'')]\!]$ and $(S_2,s_2,Act_2,\to_2) = [\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}''_{|V}) \parallel P_W(\vec{\iota}''_{|W})))))]\!]$. Let R be the smallest relation such that $P(\vec{\iota}')\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}'_{|V}) \parallel P_W(\vec{\iota}'_{|W})))))$ for any closed expression $\vec{\iota}' : \vec{D}$. We show that R is a strong bisimulation relation up to $\leftrightarrows$. Pick any arbitrary closed expression $\vec{\iota} : \vec{D}$ and suppose that $P(\vec{\iota})\ R\ \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})))))$ holds.

- Case $P(\vec{\iota}) \xrightarrow{\omega}_1 Q'$. There is an index $r \in I$ and a closed expression $l : E_r$ such that for $\sigma = [\vec{d} \leftarrow \vec{\iota}, e_r \leftarrow l]$ it holds that $[\![\sigma(c_r)]\!]$, $\omega = [\![\sigma(\alpha_r)]\!]$ and $Q' = P(\sigma(\vec{g}_r))$. There are three cases to consider based on the index r.
  - Case $r \in I \setminus (K_V \cup K_W)$. From SYN this means that $r \in (J_V \cap J_W)$. We derive the transitions using requirement ORI. First, from $[\![\sigma(c^V_r \wedge c^W_r)]\!]$ it follows that:

$$P_V(\vec{\iota}_{|V}) \xrightarrow{[\![\sigma(\alpha^V_r|\mathsf{sync}^V_r(\vec{h}^V_r))]\!]}_2 P_V(\sigma(\vec{g}_{r|V}))$$
$$\text{and } P_W(\vec{\iota}_{|W}) \xrightarrow{[\![\sigma(\alpha^W_r|\mathsf{sync}^W_r(\vec{h}^W_r))]\!]}_2 P_W(\sigma(\vec{g}_{r|W}))$$

  Furthermore, $[\![\sigma(\alpha_r)]\!] = [\![\sigma(\alpha^V_r|\alpha^W_r)]\!]$ and by rule PAR there is:

$$P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{sync}_V(\sigma(\vec{h}^V_r)))|\mathsf{sync}_W(\sigma(\vec{h}^W_r)))]\!]}_2 P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))$$

  From $[\![\sigma(\vec{h}^V_r)]\!] = [\![\sigma(\vec{h}^W_r)]\!]$ it follows that:

$$\theta_{H'}(\theta_H(\gamma_C([\![\sigma(\alpha_r)|\mathsf{sync}_V(\sigma(\vec{h}^V_r)))|\mathsf{sync}_W(\sigma(\vec{h}^W_r))]\!]))) = [\![\sigma(\alpha_r)]\!]$$

  From $\underline{[\![\sigma(\alpha_r)]\!]} \in A$ we know by Lemma A.3 that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\iota}_{|V}) \parallel P_W(\vec{\iota}_{|W})))))) \xrightarrow{[\![\sigma(\alpha_r)]\!]}_2 \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W})))))))$$

  Finally, $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W})))))))$.

– Case $r \in K_V$. We derive $P_V(\vec{\imath}_{|V}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{tag}]\!]}_2 P_V(\sigma(\vec{g}_{r|V}))$, because $[\![\sigma(c_r)]\!]$ holds. There is a transition $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{[\![\sigma(\alpha_r)|\mathsf{tag}]\!]}_2 P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\vec{\imath}_{|W})$ by From rule ParL. Furthermore, by definition $\theta_{H'}(\theta_H(\gamma_C(\sigma(\alpha_r)|\mathsf{tag}))) = \sigma(\alpha_r)$ and $\underline{\sigma(\alpha_r)} \in A$. From Lemma A.3 we conclude that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}))))) \xrightarrow{\sigma(\alpha_r)}_2 \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\vec{\imath}_{|W})))))$$

By IND it holds that $\vec{g}_{r|W} = \vec{\imath}_{|W}$. Finally, by definition $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.

– Case $r \in K_W$. Follows from the same observations as $r \in K_V$.

- Case $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W})))))) \xrightarrow{\omega}_2 Q'$. By Lemma A.4 there is an expression $Q \in \mathsf{S}$ such that $Q' = \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(Q))))$, and $\omega' \in \Omega$ such that $\omega = \theta_{H'}(\theta_H(\gamma_C(\omega')))$, $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 Q$ and $\underline{\omega} \in A$. There are three cases where a parallel composition results in a transition. Suppose that $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 Q$ is due to:

  – Rule Par with premise $P_V(\vec{\imath}_{|V}) \xrightarrow{\omega_V}_2 P'_V$ and $P_W(\vec{\imath}_{|W}) \xrightarrow{\omega_W}_2 P'_W$. Then there is a transition $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega_V+\omega_W}_2 P'_V \parallel P'_W$ such that $\omega_V + \omega_W = \omega'$ and $Q = P'_V \parallel P'_W$.
  From $\underline{\theta_{H'}(\theta_H(\gamma_C(\omega_V + \omega_W)))} \in A$ we know that $\gamma_C(\omega_V + \omega_W) = \omega + \wr \mathsf{sync}_r \wr$, for some index $r \in I$, because only a single tag is allowed with original action labels. Therefore, it also follows that $r \in I \setminus (K_V \cup K_W)$.
  Observe that the sets of indices V and V^c are disjoint (and similarly W and W^c). Also $\vec{d}_{|V}$ are the parameters of P_V and $\vec{d}_{|V^c}$ are sum variables in the r-indexed summand for which any value can be chosen. Therefore, by the existence of these transitions in the premise there are closed expressions $l, l' : E_r$, and $\vec{\kappa} : \vec{D}_{|V^c}$, $\vec{\kappa}' : \vec{D}_{|W^c}$ with the substitutions

$$\sigma = [\vec{d}_{|V} \leftarrow \vec{\imath}_{|V}, \vec{d}_{|V^c} \leftarrow \vec{\kappa}, e_r \leftarrow l]$$
$$\text{and } \sigma' = [\vec{d}_{|W} \leftarrow \vec{\imath}_{|W}, \vec{d}_{|W^c} \leftarrow \vec{\kappa}', e_r \leftarrow l']$$

  such that $[\![\sigma(c_r^V)]\!]$ holds, ω_V is equal to $[\![\sigma(\alpha_r^V)|\mathsf{sync}_r^V(\vec{h}_r^V))]\!]$ and $P'_V = P_V([\![\sigma(\vec{g}_{r|V})]\!])$. And $[\![\sigma'(c_r^W)]\!]$ holds, $\omega_W = [\![\sigma'(\alpha_r^W)|\mathsf{sync}_r^W(\vec{h}_r^W)]\!])$ and $P'_W = P_W([\![\sigma'(\vec{g}_{r|W})]\!])$ and finally $[\![\sigma(\vec{h}_r^V)]\!] = [\![\sigma'(\vec{h}_r^W)]\!]$ holds due to the synchronisation.
  From the requirement COM it follows that there is a closed expression $l'' : E_r$ such that for $\rho = [\vec{d} \leftarrow \vec{\imath}, e_r \leftarrow l'']$ that $[\![\rho(c_r)]\!]$ holds and $[\![\sigma(\alpha_r^V)|\sigma'(\alpha_r^W)]\!] = [\![\rho(\alpha_r)]\!]$. We conclude that $P(\vec{\imath}) \xrightarrow{\omega}_1 P(\rho(\vec{g}_r))$. Furthermore, from $[\![\sigma(\vec{g}_{r|V})]\!] = [\![\rho(\vec{g}_{r|V})]\!]$, and $[\![\sigma'(\vec{g}_{r|W})]\!] = [\![\rho(\vec{g}_{r|W})]\!]$ and Lemma 2.6 it follows that:

$$P_V(\sigma(\vec{g}_{r|V})) \leftrightarrows P_V(\rho(\vec{g}_{r|V}))$$
$$\text{and } P_W(\sigma'(\vec{g}_{r|W})) \leftrightarrows P_W(\rho(\vec{g}_{r|W}))$$

  By the congruence of strong bisimilarity with respect to S we obtain that:

$$\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma'(\vec{g}_{r|W})))))) \leftrightarrows \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W})))))$$

  Finally, $P(\rho(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\rho(\vec{g}_{r|V})) \parallel P_W(\rho(\vec{g}_{r|W}))))))$.

– Rule PARL with premise $P_V(\vec{\imath}_{|V}) \xrightarrow{\omega'}_2 P'_V$. Then there is a transition $P_V(\vec{\imath}_{|V}) \parallel P_W(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 P'_V \parallel P_W(\vec{\imath}_{|W})$ and $Q = P'_V \parallel P_W(\vec{\imath}_{|W})$. Pick an arbitrary index $r \in J_V$. If $r \in J_V \setminus K_V$ then the action expression contains an action labelled sync^V_r, which means that $\underline{\theta_{H'}(\theta_H(\gamma_C(\omega')))} \notin A$. Contradiction.

As such $r \in K_V$ and the requirement $\mathsf{FV}(c_r) \cup \mathsf{FV}(\alpha_r) \cup \mathsf{FV}(\vec{g}_{r|U}) \subseteq \mathsf{Vars}(\vec{d}_{|U}) \cup \{e_r\}$ holds. Therefore, there is a closed expression $l : E_r$ such that for $\sigma = [\vec{d}_{|V} \leftarrow \vec{\imath}_{|V}, e_r \leftarrow l]$ such that $[\![\sigma(c_r)]\!]$, $\omega' = [\![\sigma(\alpha_r)|\mathsf{tag}]\!]$ and $P'_V = P_V(\sigma(\vec{g}_{r|V}))$.

We know that for $\sigma' = [\vec{d} \leftarrow \vec{\imath}, e_r \leftarrow l]$ that (syntactically) $\sigma(c_r) = \sigma'(c_r)$ and $\sigma(\alpha_r) = \sigma'(\alpha_r)$. Therefore, $[\![\sigma'(c_r)]\!]$ holds and $\omega' = [\![\sigma'(\alpha_r)]\!]$ such that $P(\vec{\imath}) \xrightarrow{\omega'}_1 P(\sigma'(\vec{g}_r))$. From $\vec{g}_{r|W} = \vec{\imath}_{|W}$ (IND) we conclude that $P(\vec{\imath}) \xrightarrow{\omega'}_1 P(\sigma(\vec{g}_r))$. Finally, we conclude $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\sigma(\vec{g}_{r|V})) \parallel P_W(\sigma(\vec{g}_{r|W}))))))$.

– Rule PARR with premise $P_W(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 P'_W$. along the same lines as above.

Using Proposition A.2 we conclude that $[\![P(\vec{\imath}''_{|V})]\!] \leftrightarrows [\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V(\vec{\imath}''_{|V}) \parallel P_W(\vec{\imath}''_{|W})))))]\!]$. □

B Proof of Theorem 5.3

Theorem 5.3. Let $P(\vec{d} : \vec{D}) = +_{i\in I} \sum_{e_i:E_i} c_i \to \alpha_i \,.\, P(\vec{g}_i)$ be an LPE and $(V, K_V, J_V, c^V, \alpha^V, h^V)$ and $(W, K_W, J_W, c^W, \alpha^W, h^W)$ be separations tuples as defined in Definition 4.2. Let ψ be a state invariant of P. Given closed expressions $\vec{\imath} : \vec{D}$ such that $[\![[\vec{d} \leftarrow \vec{\imath}](\psi)]\!]$ holds the following expression, where $C = J_V \cap J_W$, is a valid decomposition:

$$\tau_{\{\mathsf{tag}\}}(\nabla_{\{\underline{\alpha_i} \mid i\in I\}\cup\{\underline{\alpha_i|\mathsf{tag}} \mid i\in(Ind_V\cup K_W)\}}(\tau_{\{\mathsf{sync}^i \mid i\in I\}}(\Gamma_{\{\mathsf{sync}^i_V \mid \mathsf{sync}^i_W \to \mathsf{sync}^i \mid i\in I\}}(P_V^{\psi,C}(\vec{\imath}_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}_{|W}))))))$$

Proof. Let ψ be a state invariant of P and let $\vec{\imath}'' : \vec{D}$ be a closed expression such that $[\![[\vec{d} \leftarrow \vec{\imath}''](\psi)]\!]$ holds. Let $(S_1, s_1, Act_1, \to_1) = [\![P(\vec{\imath}'')]\!]$ and $(S_2, s_2, Act_2, \to_2) = [\![\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\vec{\imath}''_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}''_{|W})))))]\!]$.

Let R be the smallest relation such that $P(\vec{\imath}')\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\vec{\imath}'_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}'_{|W})))))$, for any closed expression $\vec{\imath}' : \vec{D}$ such that $[\![[\vec{d} \leftarrow \vec{\imath}'](\psi)]\!]$ holds. We show that R is a strong bisimulation relation up to $\leftrightarrows$. The rest of the proof follows the same structure as the proof presented in Appendix A.

Consider the case $P(\vec{\imath}) \xrightarrow{\omega}_1 Q'$. First of all, we know that $[\![[\vec{d} \leftarrow \vec{\imath}](\psi)]\!]$ holds by definition of R. In the case $r \in I \setminus (K_V \cup K_W)$, which means that $r \in (J_V \cap J_W)$, we can therefore conclude that $[\![\sigma(c^V_r \wedge c^W_r \wedge \psi)]\!]$ holds. Furthermore, by definition of a state invariant we know that $[\![[\vec{d} \leftarrow \sigma(\vec{g}_r)](\psi)]\!]$ and thus $P(\sigma(\vec{g}_r))\, R\, \tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\sigma(\vec{g}_{r|V})) \parallel P_W^{\psi,C}(\sigma(\vec{g}_{r|W}))))))$. The other case deal with unrestricted summands and as such only the observation that $[\![[\vec{d} \leftarrow \sigma(\vec{g}_r)](\psi)]\!]$ holds needs to be added.

Consider the case $\tau_{H'}(\nabla_A(\tau_H(\Gamma_C(P_V^{\psi,C}(\vec{\imath}_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}_{|W})))))) \xrightarrow{\omega}_2 Q'$. We can easily see that the restricted condition imply the original condition as well. In the first case $P_V^{\psi,C}(\vec{\imath}_{|V}) \xrightarrow{\omega_V}_2 P'_V$ and $P_W^{\psi,C}(\vec{\imath}_{|W}) \xrightarrow{\omega_W}_2 P'_W$ and rule PAR we observe that if $[\![\sigma(c^V_r \wedge \psi)]\!]$ holds then $[\![\sigma(c^V_r)]\!]$ holds as well, and similarly if $[\![\sigma'(c^W_r \wedge \psi)]\!]$ holds that $[\![\sigma'(c^W_r)]\!]$ holds. The remainder of the proof stays exactly the same. In the case $P_V^{\psi,C}(\vec{\imath}_{|V}) \xrightarrow{\omega'}_2 P'_V$ and rule PARL such that $P_V^{\psi,C}(\vec{\imath}_{|V}) \parallel P_W^{\psi,C}(\vec{\imath}_{|W}) \xrightarrow{\omega'}_2 P'_V \parallel P_W^{\psi,C}(\vec{\imath}_{|W})$ we observe that $r \in (J_V \setminus K_V)$ and as such the condition expression is not restricted. □

On Encoding Primitives for Compensation Handling as Adaptable Processes (Oral Communication)

Jovana Dedeić
University of Novi Sad, Serbia

Jovanka Pantović
University of Novi Sad, Serbia

Jorge A. Pérez
University of Groningen, The Netherlands

This note concerns the relative expressiveness of calculi for concurrency with constructs for *compensation handling* and *dynamic update*, which are important in the rigorous specification of reliable communicating systems. Compensations and updates are intuitively similar: both specify how the behavior of a concurrent system changes at runtime in response to an exceptional event. Process calculi including these constructs, however, are technically quite different.

In a recent journal paper, we develop *valid encodings* of a calculus with compensation handling into a calculus of adaptable processes. The encodings differ in the target language: the first considers adaptable processes with *subjective updates*, in which a process reconfigures itself; the second considers *objective updates*, in which a process is reconfigured by a process in its context. We established that subjective updates are more *efficient* than objective ones in encoding primitives for compensation handling: the first encoding requires less steps than the second one to mimic a single step in the source language of compensable processes.

Introduction

In service-oriented systems, *long-running transactions* (LRTs) are computing activities which extend in time and may involve distributed, loosely coupled resources [7]. These features distinguish LRTs from traditional database-like transactions. One particularly delicate aspect of LRTs management is handling (partial) failures: mechanisms for detecting failures and bringing the LRT back to a consistent state need to be explicitly programmed.

Because designing and programming such mechanisms is error prone, specialized constructs, such as *exceptions* and *compensations*, have been put forward to offer direct programming support. Our focus is in compensation mechanisms: as their name suggests, they are meant to compensate the failure or cancellation of an LRT. Upon receiving a failure signal, a compensation mechanism is expected to install and activate behaviors for recovering system consistency. Such a compensation behavior may be different from the LRT's initial behavior.

Several calculi for concurrency with constructs for compensation handling has been proposed (see, e.g., [12, 3, 7]). Building upon process calculi such as the π-calculus [13], they capture different forms of error recovery and offer reasoning techniques on communicating processes with compensation constructs. Some works have compared the expressiveness of these calculi [3, 2, 10, 11]. In particular, Lanese et al. [10] address this question by considering a basic process language on top of which different primitives for error handling, distilled from an extensive literature, can be uniformly considered. Their core calculus of compensable processes extends the π-calculus with: *transactions* $t[P,Q]$, *protected block* $\langle P \rangle$, and *compensation update* $\mathtt{inst}\lfloor \lambda Y.R \rfloor.P$. If compensations do not allow compensation updates it calls *static* recovery, otherwise *dynamic* recovery. In transactions $t[P,Q]$ processes P and Q represent default and compensation activities, respectively. Transactions can be nested: process P in $t[P,Q]$ may contain other transactions. Also, they can be cancelled: process $t[P,Q]$ behaves as P until an *error notification* (failure signal) arrives along name t. Error notifications are output messages coming from inside

To appear in EPTCS.

or outside the transaction. To illustrate this the simplest manifestation of compensations, consider the following transitions:

$$t[P,Q] \mid \bar{t}.R \xrightarrow{\tau} Q \mid R \qquad\qquad t[\bar{t}.P_1 \mid P_2,Q] \xrightarrow{\tau} Q \tag{1}$$

The left (resp. right) transition shows how t can be canceled by an external (resp. internal) signal. Failure discards the default behavior; the compensation activity is executed instead. In both cases, the default activity (i.e., P and P_1,P_2) are entirely discarded. This may not be desirable in all cases; after a compensation is enabled, we may like to preserve (some of) the behavior in the default activity. To this end, one can use *protected blocks* to shield a process from failure signals. These blocks are transparent: Q and $\langle Q\rangle$ have the same behavior, but $\langle Q\rangle$ is not affected by failure signals. This way, the transition

$$t_2[P_2,Q_2] \mid \overline{t_2} \xrightarrow{\tau} \langle Q_2\rangle,$$

says that the compensation behavior Q_2 will be immune to failures. In the following example, we concentrate on static recovery and the response to failures is specified via *discarding*, *preserving*, and *aborting* semantics. Let us consider the process: $P = t\big[t_1[P_1,Q_1] \mid t_2[\langle P_2\rangle,Q_2] \mid R \mid \langle P_3\rangle,Q_5\big]$. We then have:

$$\begin{array}{lll}
\text{Discarding semantics:} & \bar{t} \mid P & \xrightarrow{\tau}_{\mathtt{D}} \ \langle P_3\rangle \mid \langle Q_5\rangle;\\
\text{Preserving semantics:} & \bar{t} \mid P & \xrightarrow{\tau}_{\mathtt{P}} \ \langle P_3\rangle \mid \langle Q_5\rangle \mid t_1[P_1,Q_1] \mid t_2[\langle P_2\rangle,Q_2];\\
\text{Aborting semantics:} & \bar{t} \mid P & \xrightarrow{\tau}_{\mathtt{A}} \ \langle P_3\rangle \mid \langle Q_5\rangle \mid \langle P_2\rangle \mid \langle Q_1\rangle \mid \langle Q_2\rangle.
\end{array}$$

Thus, the three different semantics implement different levels of protection. In the *discarding* semantics only top-level protected blocks are preserved. Therefore, it only concerns the compensation activity for transaction t and the protected block $\langle P_3\rangle$. The preserving semantics protects also the nested transactions t_1 and t_2; a process such as R, without an enclosing protected block, is discarded. Finally, the aborting semantics preserves all protected blocks and compensation activities in the default activity for t, including those in nested transactions, such as $\langle P_2\rangle$.

When compensation updates are allowed compensations admit *dynamic* recovery. Compensation update is performed by a new operator $\mathtt{inst}\lfloor\lambda Y.R\rfloor.P'$, where function $\lambda Y.R$ is the compensation update (Y can occur inside R). Applying such a compensation update to compensation Q produces a new compensation $R\{Q/Y\}$ after internal transition. Note that R may not occur at all in the resulting compensation, and it may also occur more than once. For instance, $\lambda Y.\mathbf{0}$ deletes the current compensation. One key merit of this approach is that different proposals arise as instances. As in static recovery, the response to failures can be specified via *discarding*, *preserving*, and *aborting* semantics.

The language in [10] leads to six distinct calculi with compensation primitives.

In a somewhat different vein, a calculus of *adaptable processes* was proposed by Bravetti et al. [1] to specify *dynamic update* in communicating systems. Adaptable processes can express forms of reconfiguration that are triggered by exceptional events, not necessarily catastrophic. An adaptable process can be deployed in a *location*, which serves as delimiters for dynamic updates. A process P *located at* l, denoted $l[P]$, can be reconfigured by an *update prefix* $l\{(X).Q\}.R$, where Q denotes an adaptation routine for l, parameterized by variable X. With these two constructs, dynamic update is realized by the following reduction rule, in which C_1 and C_2 denote contexts of arbitrarily nested locations:

$$C_1\big[l[P]\big] \mid C_2\big[l\{(X).Q\}.R\big] \longrightarrow C_1\big[Q\{P/X\}\big] \mid C_2\big[R\big] \tag{2}$$

We call this an ***objective update***: a located process is reconfigured in its own context by an update prefix at a different context: the update prefix $l\{(X).Q\}$ *moves* from C_2 to C_1, and the reconfigured behavior

$Q\{^P/_X\}$ is left in C_1. Notice that X may occur zero or many times in Q; if Q does not contain X then the current behavior P will be erased as a result of the update. This way, dynamic update is then a form of process mobility, implemented using *higher-order process communication* as found in languages such as, e.g., the higher-order π-calculus [14], the Kell calculus [15], and Homer [9].

An alternative to objective update is ***subjective update***, where process reconfiguration flows in the opposite direction:

$$C_1\left[l[P] \mid R_1\right] \mid C_2\left[l\{(X).Q\}.R\right] \longrightarrow C_1\left[\mathbf{0} \mid R_1\right] \mid C_2\left[Q\{^P/_X\} \mid R\right] \tag{3}$$

As objective update, subjective update relies on process mobility; however, the direction of movement is different: above, process P moves from C_1 to C_2, and the reconfigured behavior $Q\{^P/_X\}$ is left in C_2, not in C_1. Thus, in a subjective update the located process "reconfigures itself", which makes for a more autonomous semantics for adaptation than objective updates.

Recent Results

In our recent journal paper [4], we have compared process calculi with compensation handling (as formalized in [10]) and with dynamic update (as formalized in [1]), from the point of view of *relative expressiveness*. There are good reasons for focusing on the formal models in [10, 1]. On the one hand, the calculus of compensable processes in [10] is expressive enough to uniformly capture several different languages proposed in the literature. Because of its expressiveness, this calculus provides an appropriate starting point for further investigations. On the other hand, the calculus of adaptable processes in [1] is a minimal process model of dynamic reconfiguration, based on a few process constructs and endowed with a simple operational semantics, which can support both objective and subjective updates. Our preliminary studies [5, 6] suggest that adaptable processes provide a flexible framework to elucidate the underpinnings of compensation handling from a fresh perspective.

In this context, our journal paper [4] presents the following contributions:

1. We develop two *translations* of a core calculus with compensation handling with discarding semantics [10] into adaptable processes [1]: the first translation relies on objective updates; the second exploits subjective updates.
2. We establish that the two language translations are *valid encodings* [8], i.e., they satisfy structural properties (compositionality and name invariance) and semantic properties (operational correspondence, divergence reflection, and success sensitiveness) that bear witness to their robustness.
3. We exploit the correctness properties of our encodings to distinguish between subjective and objective updates in calculi for concurrency. We introduce an encodability criterion called *efficiency*, which allows us to formally state that subjective updates are better suited to encode compensation handling than objective updates, because they induce tighter operational correspondences.

We briefly elaborate on points (1) and (3):

- Concerning (1), while we focus on compensable processes with discarding semantics, we also consider translations in which the source calculus uses preserving semantics, aborting semantics, and dynamic compensations.
- Concerning (3), our encoding into adaptable processes with objective updates reveals a limitation: in modeling the "recollection" of protected blocks scattered within nested transactions, objective updates leave behind processes in the "wrong" location. To remedy this, the encoding uses additional synchronizations to bring processes into the appropriate locations. This reflects prominently

in the *cost* of mimicking a source computation step, as measured by the number of its associated target computation steps (which are spelled out by our statements of operational correspondence). The encoding into the calculus with subjective updates does not have this limitation.

As already mentioned, the encodings presented in [4] refine and extend results first reported in [5, 6]. Below we list the key extensions of the results that are developed in our journal paper:

1. We develop the class of *well-formed* compensable processes to formalize our encodings, for which error notifications are crucial. Precisely, this class of processes disallows certain non-deterministic interactions that involve nested transactions and error notifications.
2. We extend the criteria included in the definition of valid encoding. The following criteria have been added: *name invariance*, *divergence reflection* and *success sensitiveness*. Therefore, we included additional definitions and theorems that establish that our encoding satisfies these new criteria.
3. We develop additional definitions and theorems necessary to complete the proof of operational correspondence (completeness and soundness).

Conclusion

The recent journal paper [4] addresses programming abstractions for compensation handling and runtime adaptation by analyzing the relative expressiveness of existing calculi. We believe that a presentation based on [4] is in the scope of ICE workshop for two reasons. First, formal models for concurrency with compensation and adaptation, have been intensively studied, from multiple angles, by researchers in the ICE community. Second, the technical results in [4] can be convincingly presented to a broad audience by means of compelling examples.

References

[1] M. Bravetti, C. D. Giusto, J. A. Pérez, and G. Zavattaro. Adaptable processes. *Logical Methods in Computer Science*, 8(4), 2012.

[2] M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.

[3] L. Caires, C. Ferreira, and H. T. Vieira. A process calculus analysis of compensations. In *Proc. of TGC 2008*, volume 5474 of *LNCS*, pages 87–103. Springer, 2009.

[4] J. Dedeić, J. Pantović, and J. A. Pérez. On primitives for compensation handling as adaptable processes. *Journal of Logical and Algebraic Methods in Programming*, page 100675, 2021.

[5] J. Dedeic, J. Pantovic, and J. A. Pérez. On compensation primitives as adaptable processes. In *EXPRESS/SOS 2015*, volume 190 of *EPTCS*, pages 16–30, 2015.

[6] J. Dedeić, J. Pantović, and J. A. Pérez. Efficient compensation handling via subjective updates. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 51–58, New York, NY, USA, 2017. ACM.

[7] C. Ferreira, I. Lanese, A. Ravara, H. T. Vieira, and G. Zavattaro. Advanced mechanisms for service combination and transactions. In *Results of SENSORIA*, volume 6582 of *LNCS*, pages 302–325. Springer, 2011.

[8] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010.

[9] T. T. Hildebrandt, J. C. Godskesen, and M. Bundgaard. Bisimulation congruences for Homer - a calculus of higher-order mobile embedded resources. Technical Report TR-2004-52, IT University, 2004.

[10] I. Lanese, C. Vaz, and C. Ferreira. On the expressive power of primitives for compensation handling. In *Proc. of ESOP 2010*, volume 6012 of *LNCS*, pages 366–386. Springer, 2010.

[11] I. Lanese and G. Zavattaro. Decidability results for dynamic installation of compensation handlers. In *COORDINATION*, volume 7890 of *LNCS*, pages 136–150. Springer, 2013.

[12] C. Laneve and G. Zavattaro. Foundations of web transactions. In *Proc. of FOSSACS 2005*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005.

[13] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.

[14] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[15] A. Schmitt and J. Stefani. The Kell calculus: A family of higher-order distributed process calculi. In C. Priami and P. Quaglia, editors, *Global Computing, IST/FET International Workshop, GC 2004, Rovereto, Italy, March 9-12, 2004, Revised Selected Papers*, volume 3267 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2004.

Processes, Systems & Tests: Defining Contextual Equivalences

Clément Aubert
School of Computer and Cyber Sciences, Augusta University, Georgia, USA
caubert@augusta.edu

Daniele Varacca
LACL, Université Paris-Est Créteil, France
daniele.varacca@u-pec.fr

In this position paper, we would like to offer and defend a template to study equivalences between programs—in the particular framework of process algebras for concurrent computation. We believe that our layered model of development will clarify the distinction that is too often left implicit between the tasks and duties of the programmer and of the tester. It will also enlighten pre-existing issues that have been running across process algebras such as the calculus of communicating systems, the π-calculus—also in its distributed version—or mobile ambients. Our distinction starts by subdividing the notion of process in three conceptually separated entities, that we call *Processes*, *Systems* and *Tests*. While the role of what can be observed and the subtleties in the definitions of congruences have been intensively studied, the fact that *not every process can be tested*, and that *the tester should have access to a different set of tools than the programmer* is curiously left out, or at least not often formally discussed. We argue that this blind spot comes from the under-specification of contexts—environments in which comparisons occur—that play multiple distinct roles but supposedly always "stay the same".

1 Introduction

In the study of programming languages, contextual equivalences play a central role: to study the behavior of a program, or a process, one needs to observe its interactions with different environments, e.g. what outcomes it produces. If the program is represented by a term in a given syntax, environments are often represented as contexts surrounding the terms. But contexts play multiple roles that serve different actors with different purposes. The programmer uses them to construct larger programs, the user employs them to provide input and obtain an output, and the tester or attacker uses them to debug and compare the program or to try to disrupt its intended behavior.

We believe that representing those different purposes with the same "monolithic" syntactical notion of context forced numerous authors to repeatedly "adjust" their definition of context without always acknowledging it. We also argue that collapsing multiple notions of contexts into one prevented further progress. In this article, we propose a way of clarifying how to define contextual equivalences, and show that having co-existing notions of equivalences legitimates and explains recurring choices, and supports a rigorous guideline to separate the development of a program from its usage and testing.

Maybe in the mind of most of the experts in the formal study of programming language is our proposal too obvious to discuss. However, if this is the case, we believe that this "folklore" remains unwritten, and that since we were not at *that* "seminar at Columbia in 1976"[1], we are to remain in darkness.

We believe the interactive and friendly community of ICE to be the ideal place to reach curious and open-minded actors in the field, and to either be proven wrong or—hopefully—impact some researchers.

[1] To re-use in our setting Paul Taylor's witty comment [76].

Submitted to:
ICE 2021

Non-technical articles can at times have a tremendous impact [25], and even if we do not claim to have Dijkstra's influence or talent, we believe that our precise, documented proposition can shed a new light on past results, and frame current reflections and future developments. We also believe that ignoring or downplaying the distinctions we stress have repeatedly caused confusions.

2 The Flow of Testing

We begin by illustrating with a simple Java example the three syntactic notions–*process*, *system* and *test*–we will be using. Imagine giving a user the code `while(i < 10){x *= x; i++;}`. A user cannot execute or use this "snippet" unless it is *wrapped* into a method, with adequate header, and possibly variable declaration(s) and `return` statement. Once the programmer performed this operation, the user can *use* the obtained program, and the tester can *interact* with it further, e.g. by calling it from a `main` method.

All in all, a programmer would build on the snippet, then the tester would build an environment to interact with the resulting program, and we could obtain the code below. Other situations could arise–e.g., if the snippet was already wrapped–, but we believe this is a fair rendering of "the life of a snippet".

```
public class Main{
  public static int foo(int x){
    int i = 0;
    while(i < 10){
      x *= x;
      i++;
    }          // Snippet
    return x;
  }                    // Wrapping
  public static void main(){
    System.out.print(foo(2));
  }
}                              // Interaction
```

In this example, the snippet is what we will call a *process*, the snippet once wrapped is what we will call a *system* and the "Interaction" part without the system in it, but with additional "observations"—i.e. measures on the execution, terminal output—, is what we will call a *test*. Our terminology comes from the study of concurrent process algebras, where most of our intuitions and references are located, but let us first make a brief detour to examine how our lens applies to λ-calculus.

3 A Foreword on λ-Calculus

Theoretical languages often take λ-calculus as a model or a comparison basis. It is often said that the λ-calculus is to sequential programs what the π-calculus is to concurrent programs [69,78]. Indeed, pure λ-calculus (i.e. without types or additional features like probabilistic sum [27] or quantum capacities [74,77]) is a reasonable [6], Turing-complete and elegant language, that requires only a couple of operators, one reduction rule and one equivalence relation to produce a rich and meaningful theory, sometimes seen as an idealized target language for functional programming languages.

Since most terms[2] do not reduce as they are, to study their behavior, one needs first to make them interact with an environment, represented by a context. Contexts are generally defined as "term[s] with some holes" [11, p. 29, 2.1.18], that we prefer to call *slots* and denote $[\Box]$. Under this apparent simplicity, they should not be manipulated carelessly, as having multiple slots or not being careful when defining what it means to "fill a slot" can lead to e.g. lose confluence [16, pp. 40–41, Example 2.2.1], and as those issues persist even in the presence of a typing system [33]. Furthermore, definitions and theorems that use contexts frequently impose some restrictions on the contexts considered, to exclude e.g. $(\lambda x.y)[\Box]$ that

[2] Actually, if application, abstraction and variables all count as one, the ratio between normal term and term with redexes is unknown [15]. We imply here "since *most interesting* terms", in the sense of terms that represent programs.

simply "throw away" the term put in the slot in one step of reduction. Following the above observations, we conclude that contexts often come in two flavors, depending on the nature of the term considered:

For closed terms (i.e. without free variables), a context is essentially a series of arguments to feed the term. This observation allows to define e.g. *solvable terms* [11, p. 171, 8.3.1 and p. 416, 16.2.1].

For open terms (i.e. with free variables), a context is a *Böhm transformation* [11, p. 246, 10.3.3], which is equivalent [11, p. 246, 10.3.4] to a series of abstractions followed by a series of applications, and sometimes called "head context" [7, p. 25].

Being closed corresponds to being "wrapped"–ready to use–, and feeding arguments to a term corresponds to interacting with it from a `main` method: the Böhm transformation actually encapsulates two operations at once. In this case, the interaction can observe different aspects: whether the term terminates, whether it grows in size, etc., but it is generally agreed upon that no additional operator or reduction rule should be used. Actually, the syntax is restricted when testing, as only application is allowed: the tested term should not be wrapped in additional layers of abstraction if it is already closed.

Adding features to the λ-calculus certainly does not restore the supposed purity or unicity of the concept of context, but actually distances it even further from being simply "a term with a slot". For instance, contexts are narrowed down to term context [77, p. 1126] and surface context [27, pp. 4, 10] for respectively quantum and probabilistic λ-calculus, to "tame" the power of contexts. In resource sensitive extensions of the λ-calculus, the quest for full abstraction even led to a drastic separation of λ-terms between terms and tests [20], a separation naturally extended to contexts [19, p. 73, Figure 2.4].

This variety happened after the 2000's formal studies of contexts was undertaken [16, 17, 33], which led to the observation that treating contexts "merely as a notation [...] hinders any formal reasoning[, while treating them] as first-class objects [allows] to gain control over variable capturing and, more generally, 'communication' between a context and expressions to be put into its holes" [17, p. 29]. It is ironic that λ-calculists took inspiration from a concurrent language to split their syntax in two right at its core [20, p. 97], or to study formally the *communication* between a context and the term in its slot, while concurrent languages sometimes tried to keep the "purity" and indistinguishability of their contexts.

In the case of concurrent calculi like the calculus of communicating systems (CCS) or the π-calculus, interactions with environments are also represented using a notion of context. But the status of contexts in concurrent calculi is even more unsettling when one notes that, while "wrapping" contexts are of interest mainly for open terms in lambda calculus, *all* terms need a pertinent notion of context in concurrent systems to be tested and observed. Indeed, as the notion of "feeding arguments to a concurrent process" concurs with the idea of wrapping it into a larger process, it seems that the distinction we just made between two kinds of contexts in λ-calculus cannot be ported to concurrent calculi. Our contribution starts by questioning whenever, indeed, process calculi have treated contexts as a uniform notion independently from the nature of the term or what it was used for.

4 Contextual Relations

Comparing terms is at the core of the study of programming languages, and process algebra is no exception. Generally, and similarly to what is done in λ-calculus, a comparison is deemed of interest only if it is valid in every possible context[3] an idea formally captured by the notion of (pre-)congruence. An equivalence relation $\mathscr{R}$ is usually said to be a congruence if it is closed by context, i.e. if for all P, Q

[3] As a reviewer noted, "Proving a behavioural equivalence is a congruence has a reading that is somewhat left out of the discussion in the paper. Namely, the behavioural equivalence provides a mathematical notion of behaviour and showing such

(open or closed) terms, $(P,Q) \in \mathscr{R}$ implies that for all context $C[\Box]$, $(C[P],C[Q]) \in \mathscr{R}$. (Sometimes, the additional requirement that terms in the relation needs to be similar up to uniform substitution is added [38], and sometimes [62, p. 516, Definition 2], only the closure by substitution—seen as a particular kind of context—is required.)

A notable example of congurence is *barbed congruence* [45, Definition 2.1.4, 56, Definition 8], which closes by context a reduction-closed relation used to observe "barbs"–the channel(s) on which a process can emit or receive. It is often taken to be *the* "reference behavioural equivalence" [45, p. 4], as it observes the interface of processes, i.e. on which channels they can interact over the time and in parallel.

But behind this apparent uniformity in the definition of congruences, the definition of contextual relations itself have often been tweaked by altering the definition of context, with no clear explanation nor justification, as we illustrate below.

In the calculus of communicating systems, notions as central as contextual bisimulation [8, pp. 223-224, Definition 421] and barbed equivalence [8, p. 224, Definition 424] considers only *static* contexts [8, p. 223, Definition 420], which are composed only of parallel composition with arbitrary term and restriction. As the author of those notes puts it himself, "the rules of the bisimulation game may be hard to justify [and] contextual bisimulation [...] is more natural" [8, p. 227]. But there is no justification—other than technical, i.e. because they "they persist after a transition" [8, p. 223]—as to *why* one should consider only some contexts in defining contextual equivalences.

In the π-calculus, contexts are defined liberally [72, p. 19, Definition 1.2.1], but still exclude contexts like e.g. $[\Box]+0$ right from the beginning. Congruences are then defined using this notion of context [72, p. 19, Definition 1.2.2], and strong barbed congruence is no exception [72, p. 59, Definition 2.1.17]. Other notions, like strong barbed equivalence [72, p. 62, Definition 2.1.20], are shown to be a non-input congruence [72, p. 63, Lemma 2.1.24], which is a notion relying on contexts that forbids the slot to occur under an input prefix [72, p. 62, Definition 2.1.22]. In other words, two notions of contexts and of congruences co-exist generally in π-calculus, but "[i]t is difficult to give rational arguments as to why one of these relations is more reasonable than the other." [34, p. 245]

In the distributed π-calculus, contexts are restricted right from the beginning to particular operators [34, Definition 2.6]. Then, relations are defined to be contextual if they are preserved by static contexts [34, Definition 2.6], which contains only parallel composition with arbitrary terms and name binding. These contexts also appear as "configuration context" [41, p. 375] or "harness" in the ambient calculus [32, p. 372]. Static operators are deemed "sufficient for our purpose" [34, p. 37] and static contexts only are considered "[t]o keep life simple" [34, p. 38], but no further justification is given.

In the semantic theory for processes, at least in the foundational theory we would like to discuss below, one difficulty is that the class of formal theories restricted to "reduction contexts" [38, p. 448] still fall short on providing a satisfactory "formulation of semantic theories for processes which does not rely on the notion of observables or convergence". Hence, the authors have to furthermore restrict the class of terms to *insensitive* terms [38, p. 450] to obtain a notion of *generic reduction* [38, p. 451] that allows a satisfactory definition of sound theories [38, p. 452]. Insensitive terms are essentially the collection of terms that do not interact with contexts [38, p. 451, Proposition 3.15], an analogue to λ-calculuus' genericity Lemma [11, p. 374, Proposition 14.3.24]. Here, contexts are restricted by duality: insensitive terms are terms that will *not* interact with the context in which they are placed, and that need to be equated by sound theories.

---

behaviour is preserved by language contexts attests that the latter are proper functions of behaviour (i.e., if $C[\cdot]$ is provided with 'behaviour' then the result is still 'behaviour'). In this sense, the congruence result can be viewed as a sanity check on the language constructors."

Across calculi, a notion of "closing context"—that emerged from λ-calculus [8, p. 85], and matches the "wrapping" of a snippet—can be found in e.g. typed versions of the π-calculus [72, p. 479], in mobile ambient [78, p. 134], in the applied π-calculus [1, p. 7], and in the fusion calculus [46, p. 6]. Also known as "completing context" [67, p. 466], those contexts are parametric in a term, the idea being that such a context will "close" the term under study, making it amenable to tests and comparisons.

Let us try to extract some general principles from this short survey. It seems that contexts are 1. *in appearance* given access to the same operators than terms, 2. sometimes deemed to be "un-reasonable", without always a clear justification, 3. shrunken by need, to bypass some of the difficulties they raise, or to preserve some notions, 4. sometimes picked by the term itself—typically because the same "wrapping" cannot be applied to all processes. Additionally, in all those cases, contexts are given access to a subset of operators, or restricted to contexts with particular behavior, *but never extended*. If we consider that contexts are the main tool to test the equivalence of processes, then why should the testers–or the attacker–always have access to *fewer* tools than the programmer? What reason is there not to *extend* the set of tools, of contexts, or simply take it to be orthogonal? The method we sketch below allows and actually encourages such nuances, would justify and acknowledge the restrictions we just discussed instead of adding them *passing-by*, and actually corresponds to common usage.

5 Processes, Systems and Tests

As in the λ-calculus, most concurrent calculi make a distinction between open and closed terms. For instance, the distributed π-calculus [34] implements a distinction between closed terms (called processes [34, p. 14]) and open terms, based on binding operators (input and recursion).

Most of the time, and since the origin of the calculus of communicating systems, the theory starts by considering only programs—"closed behaviour expression[s], i.e. ones with no free variable" [49, p. 73]—when comparing terms, as—exactly like in λ-calculus—they correspond to self-sufficient, well-rounded programs: it is generally agreed upon that open terms should not be released "into the wild", as they are not able to remain in control of their internal variables, and can be subject to undesirable or uncontrolled interferences. Additionally, closed terms are also the only ones to have a *reduction semantics*, which means that they can evolve without interacting with the environment–this would corresponds, in Java, to being wrapped, i.e. inserted into a proper header and ready to be used or tested.

However, in concurrent calculi, the central notions of binders and of variables have been changing, and still seem today sometimes "up in the air". For instance, in the original CCS, restriction was not a binder [49, p. 68], and by "refusing to admit channels as entities distinct from agents" [52, p. 16] and defining two different notions of scopes [52, p. 18], everything was set-up to produce a long and recurring confusion as to what a "closed" term meant in CCS. In the original definition of π-calculus [54, 55], there is no notion of closed terms, as every (input) binding on a channel introduces a new and free occurrence of a variable. However, the language they build upon—ECCS [26]—made this distinction clear, by separating channel constants and variables.

Once again in an attempt to mimic the "economy" [53, p. 86] of λ-calculus, but also taking inspiration from the claimed "monotheism" of the actor model [36], different notions such as values, variables, or channels have been united under the common terminology of "names". This is at times identified as a strength, to obtain a "richer calculus in which values of many kinds may be communicated, and in which value computations may be freely mixed with communications." [54, p. 20] However, it seems that a distinction between those notions always needs to be carefully re-introduced when discussing technically the language [8, p. 258, Remark 493], extensions to it [1, p. 4] or possible implementations [14, p. 13, 29].

Finally, let us note that extensions of π-calculus can sometimes have different binders, as e.g. output binders are binding in the private π-calculus [61, p. 113].

In the λ-calculus, being closed is what makes a term "ready to be executed in an external environment". But in concurrent calculi, being a closed term—no matter how it is defined—is often not enough, as it is routine to exclude e.g. terms with un-guarded operators like sum [72, p. 416] or recursion [52, p. 166]. However, these operators are sometimes not excluded from the start, even if they can never be parts of tested terms. The usual strategy [8, Remark 414, 52]is often to keep them "as long as possible", and to exclude them only when their power cannot be tamed any more to fit the framework or prove the desired result, such as the preservation of weak bisimulation by all contexts.

In our opinion, the right distinction is not about binders of free variables, but about the role played by the syntactic objects in the theory. As "being closed" is 1. not always well-defined, or at least changing, 2. sometimes not the only condition, we would like to use the slightly more generic adjectives *complete* and *incomplete*–wrapped or not, in our Java terminology. Process algebras generally study terms by 1. completing them if needed, 2. inserting them in an environment, 3. executing them, 4. observing them thanks to predicates on the execution ("terminates", "emitted the barb a", etc.), hence constructing equivalences, preorders or metrics [39] on them. Often, the environment is essentially made of another process composed in parallel with the one studied, and tweaked to improve the likeliness of observing a particular behavior: hence, we would like to think of them as tests that the observed systems has to pass, justifying the terminology we will be using.

Processes are "partial" programs, still under development; sometimes called "open terms", they correspond to *incomplete terms*. They would be called code fragments, or snippets, in standard programming.

Systems are "configured processes", ready to be executed in any external environment: sometimes called "closed terms", they correspond to *complete terms*. They would be functions shipped with a library in standard programming, and ready to be executed.

Tests are defined using contexts and observations, and aims at executing and testing systems. They would be `main` methods calling a library or an API in standard programming, along with a set of observables.

Our terminology is close to the one used e.g. in ADPI [34, Chapter 5] or mobile ambients [47, Table 1], which distinguish processes and systems. In the literature of process algebra, the term "process" is commonly used to denote these three layers, possibly generating confusion. We believe this usage comes from a strong desire to keep the three layers uniform, using the same name, operators and rules, but this principle is actually constantly dented (as discussed in Sect. 4), for reasons we expose below.

6 Designing Layered Concurrent Languages

Concurrent languages could benefit from this organization from their conception:

Define processes The first step is to select a set of operators called *construction operators*, used by the programmer to write processes. Those operators should be expressive, easy to combine, with constraints as light as possible, and selected with the situation that is being modeled in mind—and not depending on whenever they fare well with not-yet-defined relations, as it is often done to privilege the guarded sum over the internal choice. To ease their usage, a "meta-syntax" can be used, something that is generally represented by the structural equivalence. (Another interesting approach is proposed in "the π-calculus, at a distance" [4, p. 45], that bypasses the need for a structural equivalence without losing the flexibility it usually provides.)

Define deployment criteria How a process can become a system ready to be executed and tested should then be defined as a series of conditions on the binding of variables, the presence or absence of some construction operators at top-level, and even the addition of *deployment operators*, marking the process as ready to be deployed in an external environment[4]. Having a set of deployment operators that restricts, expands or intersects with the set of construction operators is perfectly acceptable, and it should enable the transformation of processes into systems and their composition.

Define tests The last step requires to define 1. a set of *testing operators*, 2. a notion of environment constructed from those operators, along with instructions on how to place a system in it, 3. a system of reduction rules regimenting how a system can execute in an environment, 4. a set of observables, i.e. a function from systems in environments to a subset of a set of atomic proposition (like "emits barb a", "terminates", "contains recursion operator", etc.).

Observe that each step uses its own set of operators and generates its own notion of context—to construct, to deploy, or to test. Tests would be key in defining notions of congruence, that would likely be reduction-closed, observational contextually-closed relations. Determining if tests should wrap processes into systems or if that should be done ahead of the test itself resonates with a long-standing debate in process algebra, and is discussed in Sect. 9.2. Note that compared to how concurrent languages are generally designed, our approach is refined along two axis: 1. every step previously exposed allows the introduction of novel operators, 2. multiple notions of systems or tests can and should co-exist in the same process algebra.

7 Addressing Existing Issues

In the process algebras literature, processes and systems often have the same structure as tests and all use the same operators and contexts, to preserve and nurture a supposedly required simplicity—at least on the surface of it. But actually, we believe that the distinction we offer is constantly used "under the hood", without always a clear discussion, but that it captures and clarifies some of the choices, debates, improvements and explanations that have been proposed.

Co-defining observations and contexts Originally, the barb was a predicate [56, p. 690], whose definition was purely syntactic. Probably inspired by the notion of observer for testing equivalences [23, p. 91], an alternative definition was made in terms of parallel composition with a tester process [45, p. 10, Definition 2.1.3]. This illustrates perfectly how the set of observables and the notion of context are inter-dependent, and that tests should always come with a definition of observable *and* a notion of context: we believe our proposal could help in clarifying the interplay between observations and contexts. One could even imagine having a series of "contexts and observations lemmas" illustrating how certain observations can be simulated by some operators, or reciprocally.

Justifying the "silent" transition's treatment It is routine to define relations (often called "weak") that ignore silent (a.k.a. τ-) transitions, seen as "internal". This sort of transitions was dubbed "unobservable internal activity" [34, p. 6] and sometimes opposed to "externally observable actions" [70, p. 230]. While we agree that "[t]his abstraction from internal differences is essential for any tractable theory of processes" [52, p. 3], we would also like to stress that both can and should be accommodated, and that "internal" transition should be treated as invisible *to the user*, but should still be accessible *to the programmer* when they are running their own tests.

[4] Exactly like a Java method header can use keywords—`extends`, `implements`, etc.—that cannot be used in a method body.

The question "to what extent should one identify processes differing only in their internal or silent actions?" [13, p. 6] is sometimes asked, and discussed as if it was a property of the process algebra and not something that can be *internally* tuned when needed. We argue that the answer to that question is "*it depends who is asking!*": from a user perspective, internal actions should *not* be observed, but it makes sense to let a programmer observe them when testing to help in deciding which process to prefer based on information not available to users.

Letting multiple comparisions co-exist The discussion on τ-transitions resonates with a long debate on which notion of behavioral relation is the most "reasonable", and—still recently—a textbook can conclude a brief overview of this issue by "hop[ing] that [they] have provided enough information to [their] readers so that they can draw their own conclusions on this long-standing debate" [70, p. 160]. Sometimes, a similar question is phrased in terms of choosing the right level of abstraction to obtain meaningful language comparisons [43, Section 3]. We firmly believe that the best answer to both questions is to acknowledge that different relations and comparisons tools match different needs, and that there is no "one size fits all" answer for the needs of all the variety of testers. Of course, comparing multiple relations is an interesting and needed task [28, 31], but one should also state that multiple comparison tools can and should co-exist, and such vision will be encapsulated by the division we are proposing.

Embracing a feared distinction The distinction between our notions of processes and systems is rampant in the literature, but too often feared, as if it was a parenthesis that needed to be closed to restore some supposedly required purity and uniformity of the syntax. A good example is probably given by mobile ambients [47]. The authors start with a two-level syntax that distinguishes between processes and systems [47, p. 966]. Processes have access to strictly more constructors than systems [47, p. 967, Table 1], that are supposed to hide the threads of computation [47, p. 965]. A notion of *system context* is then introduced—as a restriction of arbitrary contexts—and discussed, and two different ways for relations to be preserved by context are defined [47, p. 969, Definiton 2.2].

The authors even extend further the syntax for processes with a special $\circ$ operator [47, p. 971, Definition 3.1], and note that the equivalences studied will not consider this additional constructor: we can see at work the distinction we sketched, where operators are added and removed based on different needs, and where the language needs not to be monolithic. The authors furthermore introduce two different reduction barbed congruences [47, p. 969, Definition 2.4]—one for systems, and one for processes, with different notions of contexts—but later on prove that they coincide on systems [47, p 989, Theorem 6.10]. It seems to us that the distinction between processes and systems was essentially introduced for technical reasons, but that re-unifying the syntax—or at least prove that systems do not do more than processes—was a clear goal right from the start. We believe it would have been fruitful to embrace this distinction in a framework similar to the one we sketched: while retaining the interesting results already proven, maintaining this two-level syntax would allow to make a clearer distinction between the user's and the programmer's roles and interests, and assert that, sometimes, systems can and *should* do more than processes–for instance, interacting with users!–, and can be compared using different tools.

Keeping on extending contexts We are not the first to argue that constructors can and should be added to calculi to access better discriminatory power, but without necessarily changing the "original" language. The mismatch operator, for instance, has a similar feeling: "reasonable" testing equivalences [18, p. 280] require it, and multiple languages [2, p. 24] use it to provide finer-grained equivalences. For technical reasons [72, p. 13], this operator is generally not part of the "core" of π-calculus, but resurfaces *by need* to obtain better equivalences: we defend a liberal use of this fruitful technics, by making a clear

separation between the construction operators—added for their expressivity—and the testing operators—that improve the testing capacities.

Treating extensions as different completions It would benefit their study and usage to consider different extensions of processes algebras as different completion strategies for the same construction operators. For instance, reversible [42] or timed [80] extensions of CCS could be seen as two completion strategies—different conditions for a process to become a system—for the same class of processes, inspired from the usual CCS syntax [8, Chapter 28.1]. Those completion strategies would be suited for different needs, as one could e.g. complete a CSS process as a RCCS [22] system to test for relations such as hereditary history-preserving bisimulation [9], and then complete it with time markers as a safety-critical system. This would correspond to having multiple compilation, or deployment, strategies, based on the need, similar to "debug" and "real-time", versions of the same piece of software.We think also of Debian's DebugPackage, enabling generation of stack traces for any package, or of the `CONFIG_PREEMPT_RT` patch that converts a kernel into a real-time micro-kernel: both uses the same source code as their "casual" versions.

Obtaining fine-grained typing systems The development of typing systems for concurrent programming languages is a notoriously difficult topic. Some results in π-calculus have been solidified [72, Part III], but diverse difficulties remain. Among them, the co-existence of multiple systems for e.g. session types [35], the difficulty to tie them precisely to other type systems as Linear Logic [21], and the doubts about the adaptation of the "proof-as-program" paradigm in a concurrent setting [12], make this problem active and diverse. The ultimate goal seems to find a typing system that would accommodate different uses and scenarios that are not necessarily comparable.

Using our proposal, one could imagine easing this process by developing two different typing systems, one aimed at programmers—to track bugs and produce meaningful error messages—and one aimed at users—to track security leaks or perform user-input validation. Once again, having a system developed along the layers we recommend would allow to have e.g. a type system for processes only, and to erase the information when completing the process, so that the typing discipline would be enforced only when the program is being developed, but not executed. This is similar to arrays of parameterized types in Java [60, pp. 253–258], that checks the typing discipline at compilation time, but not at run-time.

While this series of examples and references illustrates how our proposal could clarify pre-existing distinctions, we would like to stress that 1. nothing prevents from collapsing our distinction when it is not needed, 2. additional progresses could be made using it, as we sketch in the next section.

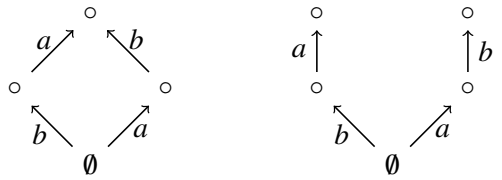
8 Exploiting Context Awareness

We would like to sketch below some possible exploitations of our frame that we believe could benefit the study and expressivity of some popular concurrent languages.

For CCS, we sketch below two possible improvements, the second being related to security.

Testing for auto-concurrency Auto-concurrency (a.k.a. auto-parallelism) is when a system has two different transitions—leading to different states—labeled with the same action [58, p. 391, Definition 5]. Systems with auto-concurrency are sometimes excluded as non-valid terms [24, p. 155] or simply not considered in particular models [59, p. 531], as the definition of bisimulation is problematic for them.

Consider e.g. the labeled configuration structures (a.k.a. stable family [79, Section 3.1]) on the right, where the label of the event executed is on the edge and configurations are represented with ∘. Non-interleaving models of concurrency [73] distinguishes between them, as "true concurrency models" would.



Some forms of "back-and-forth-bisimulations" cannot discriminate between them if $a = b$ [64]. While not being able to distinguish between those two terms may make sense from an external—user's—point of view, we argue that a programmer should have access to an internal mechanism that could answer the question "*Can this process perform two barbs with the same label at the same time?*". Such an observation—possibly coupled with a testing operator—would allow to distinguish between e.g. $!a.P \mid !a.P$ and $!a.P$, that are generally taken to be bisimilar, and would re-integrate auto-concurrent systems—that are, after all, unjustifiably excluded—in the realm of comparable systems.

Representing man-in-the-middle One could add to the testing operators an operator $\nabla a.P$, which would forbid P to act silently on channel a. This novel operator would add the possibility for the environment to "spy" on a determined channel, as if the environment was controlling (part of) the router of the tested system. One could then reduce "normally" in a context $\nabla a[\Box]$ if the channel is still secure:

$$\nabla a(b.Q \mid \bar{b}.P) \to^{\tau} \nabla a(Q \mid P) \qquad \text{(If } a \neq b\text{)}$$

But in the case where $a = b$, the environment could intercept the communication and then decide to forward, prevent, or alter it. Adding this operator to the set of testing operators would for instance open up the possibility of interpreting $\nu a(P)$ as an operation securing the channel a in P, enabling the study of relations $\sim$ that could include e.g.

$$\nabla a(\nu a(P|Q) \sim \nabla a(\nu b(P[a/b]|Q[a/b])) \qquad \text{(For } b \text{ not in the free names of } P \text{ nor } Q\text{)}$$
$$\nu a(\nabla a(P|Q)) \sim \nabla a(P|Q) \qquad \text{(Uselessness of securing a hacked channel)}$$

While the first rule enforces that, once secured, channel names are α-equivalent (the process can decide to migrate to a different channel without being spied on), the second illustrates that, once a channel is tapped, a process cannot obtain any confidentiality on it anymore.

In π-calculus, all tests are instantiating contexts (in the sense that the term tested needs to be either already closed, or to be closed by the context), and all instantiating contexts use only construction operators, and hence are "construction contexts". This situation corresponds to Situation A in Figure 1. We believe the picture could be much more general, with tests having access to *more constructors*, and not needing to be instantiating—in the sense that completion can be different from closedness—, so that we would obtain Situation B in Figure 1. While we believe this remark applies to most of the process algebras we have discussed so far, it is particularly salient in π-calculus, where the match and mismatch operators have been used "to internalize a lot of meta theory" [30, p. 57], standing "inside" the "Construction operators" circle while most authors seem to agree that they would prefer not to add it to the internals of the language[5]. It should also be noted that the mismatch operator—in its "intuitionistic" version—furthermore "tried to escape the realm of instantiating contexts" by being tightly connected [40] to *quasi-open bissimilarities* [71, p. 300, Definition 6], which is a subtle variation on how substitutions can be applied by context to the terms being tested.

[5]To be more precise: while "most occurences of matching can be encoded by parallel composition [...,] mismatching cannot be encoded in the original π-calculus" [63, p. 526], which makes it somehow suspicious.

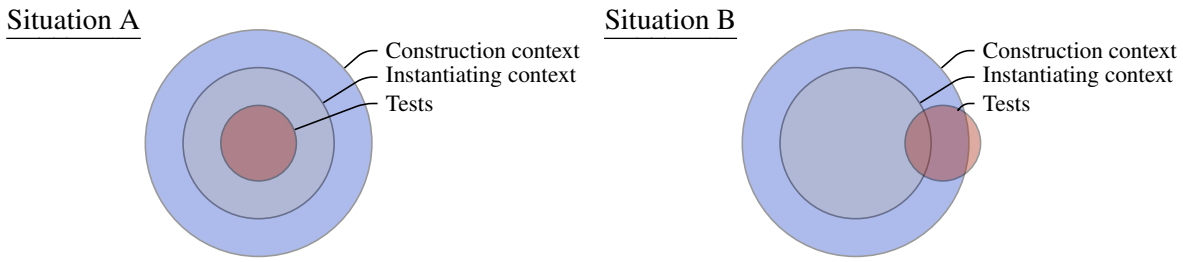


Figure 1: Opening up the testing capacities of π-calculus

Having a notion of "being complete" not requiring closedness could be useful when representing distributed programming, where "one often wants to send a piece of code to a remote site and execute it there. [...] [T]his feature will greatly enhance the expressive power of distributed programming[by] send[ing] an open term and to make the necessary binding at the remote site." [33, p. 250] We believe that maintaining the possibility of testing "partially closed"—but still complete—terms would enable a more theoretical understanding of distributed programming and remote compilation.

Distributed π-calculus, could explore the possible differences between two parallelisms: between threads in the same process—in the Unix sense—and between units of computation. Such a distinction could be rephrased thanks to two parallel operators, one on processes and the other on systems. Such a distinction would allow to observationally distinguish e.g. the execution of a program with two threads on a dual-core computer and the execution of two single thread programs on two single-core computers.

For cryptographic protocols, we could imagine representing encryption of data as a special context $\mathscr{E}[\Box]$ that would transform a process P into an encrypted system $\mathscr{E}[P]$, and make it un-executable unless "plugged" in an environment $\mathscr{D}[\Box]$ that could decrypt it. This could allow the applied π-calculus [1] to become more expressive and to be treated as a decoration of the pure π-calculus more effectively. This could also, as the authors wish, make "the formalization of attackers as contexts [...] continue to play a role in the analysis of security protocols" [1, p. 35].

Recent progresses in the field of verification of cryptographic protocols [10] hinted in this direction as well. By taking "[t]he notion of test [to] be relative to an environment" [10, p. 12], a formal development involving "frames" [10, Definition 2.3] can emerge and give flesh to some ideas expressed in our proposal. It should be noted that this work also "enrich[...] processes with a success construct" [10, p. 12], that cannot be used to construct processes, to construct "experiments".

9 Concluding Remarks

We conclude by discussing related approaches, by casting a new light on a technical issue related to barbed congruences, by offering the context lemmas a new interpretation, and by coming back to our motivations.

9.1 An Approved and Promising Perspective

We would like to stress that our proposal resonates with previous comments, and should not be treated as an isolated historical perspective that will have no impact on the future.

In the study of process algebras, in addition to the numerous hints toward our formalism that we already discussed, there are at least two instances when the power of the "testing suite" was explicitly

discussed [68, Remark 5.2.21]. In a 1981 article, it is assumed that "by varying the ambiant ('weather') conditions, an experimenter" [50, p. 32] can observe and discriminate better than a simple user could. Originally, this idea seemed to encapsulate two orthogonal dimensions: the first was that the tester could run the tested system any number of times, something that would now be represented by the addition of the replication operator ! to the set of testing operators. The second was that the tester could enumerate all possible non-deterministic transitions of the tested system. This second dimension gave birth to "a language for testing concurrent processes" [44, p. 1] that is more powerful than the language used to write the programs being tested. In this particular example, the tester has access to a termination operator and probabilistic features that are not available to the programmer: as a result, the authors "may distinguish non-bisimilar processes through testing" [44, p. 19].

Looking forward, the vibrant field of secure compilation made a clear-cut distinction between "target language contexts" representing adversarial code and programmers' "source context" to explore property preservation of programs [3]. This perspective was already partially at play in the spi calculus for cryptographic protocols [2, p. 1], where the attacker is represented as the "environment of a protocol". We believe that both approaches—coming from the secure compilation, from the concurrency community, but also from other fields—concur to the same observation that the environment—formally captured by a particular notion of context—deserves an explicit and technical study to model different interactions with systems, and need to be detached from "construction" contexts.

9.2 When Should Contexts Come into Play?

The interesting question of *when* to use contexts when testing terms [72, pp. 116–117, Section 2.4.4] raises a technical question that is put under a different perspective by our analysis. Essentially, the question is whether the congruences under study should being *defined* as congruences (e.g. reduction-closed barbed congruence [72, p. 116]), or being defined in two steps, i.e. as the contextual closure of a pre-existing relation (e.g. strong barbed congruence [72, p. 61, Definition 2.1.17], which is the contextual closure of strong barbed bisimilarity [72, p. 57, Definition 2.1.7])?

Indeed, bisimulations can be presented as an "interaction game" [75] generally played as 1. Pick an environment for both terms (i.e., complete them, then embed them in the same testing environment), 2. Have them "play" (i.e. have them try to match each other's step). But a more dynamic version of the game let picking an environment *be part of the game*, so that each process can not only pick the next step, *but also in which environment it needs to be performed*. This version of the game, called "dynamic observational congruence" [57], provides a better software modularity and reusability, as it allows to study the similarity of terms that can be re-configured "on the fly". Embedding the contexts in the definitions of the relations is a strategy that was also used to obtain behavioral characterization of theories [38, p. 455, Proposition 3.24], and that corresponds to open bisimilarities [66, p. 77, Proposition 3.12]

Those two approaches have been extensively compared and studied–still are [1, p. 24]—but to our knowledge they rarely co-exist, as if one had to take a side at the early stage of the language design, instead of letting the tester decide later on which approach is best suited for what they wish to observe. We argue that both approaches are equally valid, *provided we acknowledge they play different roles*.

This question of *when are the terms completed?* can be rephrased as *what is it that you are trying to observe?*, or even *who is completing them?*: is the completion provided by the programmer, once and for all, or is the tester allowed to explore different completions and to change them as the tests unfold? Looking back at our Java example from Sect. 2, this corresponds to letting the tester repeatedly tweak the parameter or return type of the wrapping from `int` to `long`, allowing them to have finer comparisons between snippets. In this frame, moving from the *static* definition of congruence to *dynamic* one would

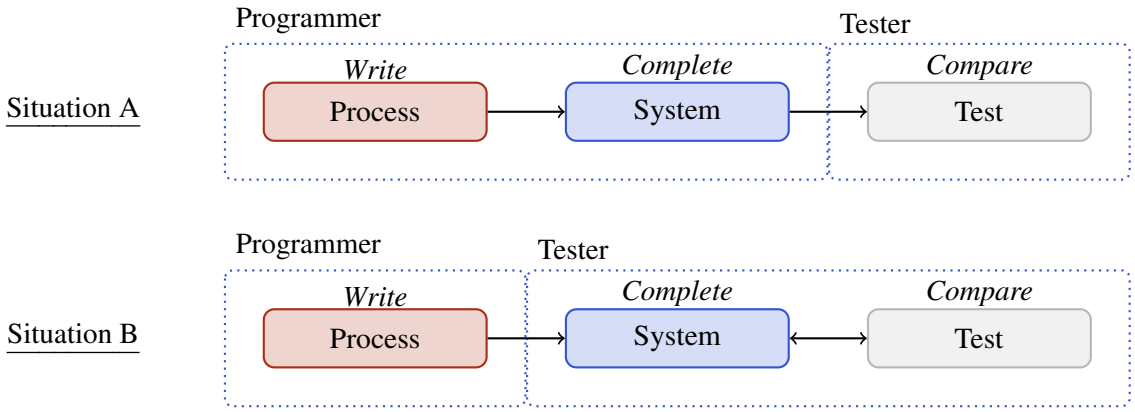


Figure 2: Distinguishing between completing strategies

corresponds to going from Situation A to Situation B in Figure 2. This illustrates two aspects worth highlighting:

1. Playing on the variation "*should I complete the terms before or during their comparison?*" is not simply a technical question, but reflects a choice between two different situations equally interesting.
2. This choice can appeal to different notions of systems, completions and tests: for instance, while completing a process before testing it (Situation A) may indeed be needed when the environment represents an external deployment platform, it makes less sense if we think of the environment as part of the development workflow, in charge of providing feedback to the programmer or as a powerful attacker than can manipulate the conditions in which the process is executed (Situation B).

If completion is seen as compilation, this opens up the possibility of studying how the bindings performed *by the user*, on *their* particular set-up, during a *remote* compilation, can alter a program. One can then compare different relations—some comparing source code's releases, some comparing binaries' releases—to get a better, fuller, picture of the program.

9.3 Penetrating Context Lemmas' Meanings

What is generally refereed to as *the* context lemma[6] is actually a series of results stating that considering all the operators when constructing the context for a congruence may not be needed. For instance, it is equivalent to define the barbed congruence [72, p. 95, Definition 2.4.5] as the closure of barbed bisimilarity under all context, or only under contexts of the form $[\Box]\sigma \mid P$ for all substitution σ and term P. In its first version [65, p. 432, Lemma 5.2.2], this lemma had additional requirements e.g. on sorting contexts, but the core idea is always the same: "*there is no need to consider all contexts to determine if a relation is a congruence, you can consider only contexts of a particular form*".

The "flip side" of the context lemma is what we would like to call the "anti-context pragmatism": whenever a particular type of operator or context prevents a relation from being a congruence, it is tempting to simply exclude it. For instance, contexts like $[\Box]+0$ are routinely removed—as we discussed in Sect. 4—to define the barbed congruence of π-calculus, or contexts were restricted to what is called harnesses in the mobile ambients calculus [32] before proving such results. As strong bisimulation [62, p. 514, Definition 1] is not preserved by input prefix [62, p. 515, Proposition 4] but is by all the other operators, it

[6] At least, in process algebra, as the same name is used for a different type of meaning in e.g. λ-calculus [48, p. 6].

is sometimes tempting to simply remove input prefix from the set of constructors allowed at top-level in contexts, which is what non-input contexts [72, p. 62, Definition 2.1.22] do, and then to establish a context lemma for this limited notion of context.

Taken together, those two remarks produce a strange impression: while it is mathematically elegant and interesting to prove that weaker conditions are enough to satisfy an interesting property, it seems to us that this result is sometimes "forced" into the process algebra by having ahead of time excluded all the operators that would not fit, hence producing a result that is not only weaker, but also somehow artificial, or even tautological. Furthermore the criteria of "not adding any discriminating power" should not be a positive criterion when deciding if a context belongs to the algebra: on the opposite, one would want contexts to *increase* the discriminating power—as for the mismatch operator—and not to "conform" to what substitution and parallel composition have already decided.

Context lemmas seem to embrace an uncanny perspective: instead of being used to prove properties about tests more easily, they should be considered from the perspective of the ease of use of testing systems. Stated differently, we believe that the set of testing operators should come first, and then *then*, if the language designer wishes to add operators to ease the testers' life, they can do so providing they obtain a context lemma proving that those operators do not alter the original testing capacities. Once again, varying the testing suite is perfectly acceptable, but once fixed, *the context lemma is simply present to show that adding some testing operators is innocent, that it will simply make testing certain properties easier*.

9.4 Embracing the Diversity

Before daring to submit a non-technical paper, we tried to conceive a technical construction that could convey our ideas. In particular we tried to build a syntactic (even categorical) meta-theory of processes, systems and tests. We wanted to define congruences in this meta-theory, and to answer the following question: what could be the minimal requirements on contexts and operators to prove a generic form of context lemma for concurrent languages?

However, as the technical work unfolded, we realized that the definitions of contexts, observations, and operators, were so deeply interwoven that it was nearly impossible to extract any general or useful principle. Context lemmas use specific features of languages, in a narrow sense, as for instance no context lemma can exist in the "Situation B" of Figure 2 [72, p. 117], and we were not able to find a unifying framework. This also suggests that context lemmas are often *fit* for particular process algebras *by chance*, and dependent intrinsically of the language considered, for no deep reasons.

This was also liberating, as all the nuances of languages we had been fighting against started to form a regular pattern: every single language we considered exhibited (at least parts of) the structure we sketched in the present proposal. Furthermore, our framework was a good lens to read and answer some of the un-spoken questions suggested in the margin or the footnotes—but rarely upfront—of the multiple research papers, lecture notes and books we consulted. So, even without mathematical proofs, we consider this contribution a good way of stirring the community, and to question the traditional wisdom.

It seems indeed to us that there is nothing but benefits in altering the notion of context, as it is actually routine to do so, even recently [37], and that stating the variations used will only improve the expressiveness of the testing capacities and the clarity of the exposition.

It is a common trope to observe the immense variety of process calculi, and to sometimes wish there could be a common formalism to capture them all—to this end, *the* π-calculus is often considered the best candidate. Acknowledging this diversity is already being one step ahead of the λ-calculus—that keeps forgetting that there is more than one λ-calculus, depending on the evaluation strategy and on features

such as sharing [5]—and this proposal encourages to push the decomposition into smaller languages even further, as well as it encourages to see whole theories as simple "completion" of standard languages. As we defended, breaking the monolithic status of context[7] will actually make the theory and presentation follow more closely the technical developments, and liberate from the goal of having to find *the* process algebra with *its unique* observation technique that would capture all possible needs.

Acknowledgements The author wish to thank the organizer of ICE 2021 for organizing this welcoming, open workshop, as well as the reviewers who kindly shared their comments, suggestions and insights with us. This paper benefited a lot from them.

[7] This may be a good place to mention that this monolithicity probably comes in part from the original will of making e.g. CCS a programming *and* specification language. The specification was supposed to be the program itself, that would be easy to check for correctness: the goal was to make it "possible to describe existing systems, to specify and program new systems, and to argue mathematically about them, all without leaving the notational framework of the calculus" [51, p. 1]. This original research project slightly shifted—from specifying programs to specifying behaviors—but that original perspective remained.

References

[1] Martín Abadi, Bruno Blanchet & Cédric Fournet (2018): *The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication*. *J. ACM* 65(1), pp. 1:1–1:41, doi:10.1145/3127586.

[2] Martín Abadi & Andrew D. Gordon (1999): *A Calculus for Cryptographic Protocols: The spi Calculus*. *Inf. Comput.* 148(1), pp. 1–70, doi:10.1006/inco.1998.2740.

[3] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani & Jérémy Thibault (2019): *Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation*. In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, IEEE, pp. 256–271, doi:10.1109/CSF.2019.00025.

[4] Beniamino Accattoli (2013): *Evaluating functions as processes*. In Rachid Echahed & Detlef Plump, editors: *TERMGRAPH 2013*, *EPTCS* 110, pp. 41–55, doi:10.4204/EPTCS.110.6.

[5] Beniamino Accattoli (2019): *A Fresh Look at the lambda-Calculus (Invited Talk)*. In Herman Geuvers, editor: *CSL*, *LIPIcs* 131, Schloss Dagstuhl, pp. 1:1–1:20, doi:10.4230/LIPIcs.FSCD.2019.1.

[6] Beniamino Accattoli & Ugo Dal Lago (2014): *Beta reduction is invariant, indeed*. In Thomas A. Henzinger & Dale Miller, editors: *CSL*, ACM, p. 8, doi:10.1145/2603088.2603105.

[7] Beniamino Accattoli & Ugo Dal Lago (2012): *On the Invariance of the Unitary Cost Model for Head Reduction*. In Ashish Tiwari, editor: *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, *LIPIcs* 15, Schloss Dagstuhl, pp. 22–37, doi:10.4230/LIPIcs.RTA.2012.22.

[8] Roberto M. Amadio (2016): *Operational methods in semantics*. Lecture notes, Université Denis Diderot Paris 7. Available at `https://hal.archives-ouvertes.fr/cel-01422101`.

[9] Clément Aubert & Ioana Cristescu (2020): *How Reversibility Can Solve Traditional Questions: The Example of Hereditary History-Preserving Bisimulation*. In Igor Konnov & Laura Kovács, editors: *CONCUR*, *LIPIcs* 2017, Schloss Dagstuhl, pp. 13:1–13:24, doi:10.4230/LIPIcs.CONCUR.2020.13.

[10] David Baelde (2021): *Contributions à la Vérification des Protocoles Cryptographiques*. Habilitation à diriger des recherches, Université Paris-Saclay. Available at `http://www.lsv.fr/~baelde/hdr/habilitation_baelde.pdf`.

[11] Hendrik Pieter Barendregt (1984): *The Lambda Calculus – Its Syntax and Semantics*. *Studies in Logic and the Foundations of Mathematics* 103, North-Holland.

[12] Emmanuel Beffara & Virgile Mogbil (2012): *Proofs as executions*. In Jos C. M. Baeten, Thomas Ball & Frank S. de Boer, editors: *IFIP TCS*, *LNCS* 7604, Springer, pp. 280–294, doi:10.1007/978-3-642-33475-7_20.

[13] Jan A. Bergstra, Alban Ponse & Scott A. Smolka, editors (2001): *Handbook of Process Algebra*. Elsevier Science, Amsterdam, doi:10.1016/B978-044482830-9/50017-5.

[14] Bruno Blanchet (2016): *Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif*. *Foundations and Trends in Privacy and Security* 1(1-2), pp. 1–135, doi:10.1561/3300000004.

[15] Olivier Bodini (2021): Personal communication.

[16] Mirna Bognar (2002): *Contexts in Lambda Calculus*. Ph.D. thesis, Vrije Universiteit Amsterdam. Available at `https://www.cs.vu.nl/en/Images/bognar_thesis_tcm210-92584.pdf`.

[17] Mirna Bognar & Roel C. de Vrijer (2001): *A Calculus of Lambda Calculus Contexts*. *J. Autom. Reasoning* 27(1), pp. 29–59, doi:10.1023/A:1010654904735.

[18] Michele Boreale & Rocco De Nicola (1995): *Testing Equivalence for Mobile Processes*. *Inf. Comput.* 120(2), pp. 279–303, doi:10.1006/inco.1995.1114.

[19] Flavien Breuvart (2015): *Dissecting denotational semantics*. Ph.D. thesis, Université Paris Diderot — Paris VII. Available at `https://www.lipn.univ-paris13.fr/~breuvart/These_breuvart.pdf`.

[20] Antonio Bucciarelli, Alberto Carraro, Thomas Ehrhard & Giulio Manzonetto (2011): *Full Abstraction for Resource Calculus with Tests*. In Marc Bezem, editor: *CSL*, *LIPIcs* 12, Schloss Dagstuhl, Dagstuhl, Germany, pp. 97–111, doi:10.4230/LIPIcs.CSL.2011.97.

[21] Luís Caires, Frank Pfenning & Bernardo Toninho (2016): *Linear logic propositions as session types*. *MSCS* 26(3), pp. 367–423, doi:10.1017/S0960129514000218.

[22] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[23] Rocco De Nicola & Matthew Hennessy (1984): *Testing Equivalences for Processes*. *Theor. Comput. Sci.* 34, pp. 83–133, doi:10.1016/0304-3975(84)90113-0.

[24] Rocco De Nicola, Ugo Montanari & Frits W. Vaandrager (1990): *Back and Forth Bisimulations*. In Jos C. M. Baeten & Jan Willem Klop, editors: *CONCUR '90*, *LNCS* 458, Springer, pp. 152–165, doi:10.1007/BFb0039058.

[25] Edsger W. Dijkstra (1968): *Letters to the editor: go to statement considered harmful*. *Commun. ACM* 11(3), pp. 147–148, doi:10.1145/362929.362947.

[26] Uffe Engberg & Mogens Nielsen (2000): *A calculus of communicating systems with label passing - ten years after*. In Gordon D. Plotkin, Colin Stirling & Mads Tofte, editors: *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, The MIT Press, pp. 599–622.

[27] Claudia Faggian & Simona Ronchi Della Rocca (2019): *Lambda Calculus and Probabilistic Computation*. In: *LICS*, IEEE, pp. 1–13, doi:10.1109/LICS.2019.8785699.

[28] Cédric Fournet & Georges Gonthier (2005): *A hierarchy of equivalences for asynchronous calculi*. *J. Log. Algebr. Methods Program.* 63(1), pp. 131–173, doi:10.1016/j.jlap.2004.01.006.

[29] Simon Fowler, Sam Lindley & Philip Wadler (2017): *Mixing Metaphors: Actors as Channels and Channels as Actors*. In Peter Müller, editor: *ECOOP 2017*, *LIPIcs* 74, Schloss Dagstuhl, pp. 11:1–11:28, doi:10.4230/LIPIcs.ECOOP.2017.11.

[30] Yuxi Fu & Zhenrong Yang (2003): *Tau laws for pi calculus*. *Theor. Comput. Sci.* 308(1-3), pp. 55–130, doi:10.1016/S0304-3975(03)00202-0.

[31] Robert J. van Glabbeek (1993): *The Linear Time - Branching Time Spectrum II*. In Eike Best, editor: *CONCUR '93*, *LNCS* 715, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.

[32] Andrew D. Gordon & Luca Cardelli (2003): *Equational Properties Of Mobile Ambients*. *MSCS* 13(3), pp. 371–408, doi:10.1017/S0960129502003742.

[33] Masatomo Hashimoto & Atsushi Ohori (2001): *A typed context calculus*. *Theor. Comput. Sci.* 266(1-2), pp. 249–272, doi:10.1016/S0304-3975(00)00174-2.

[34] Matthew Hennessy (2007): *A distributed Pi-calculus*. CUP, doi:10.1017/CBO9780511611063.

[35] Bas van den Heuvel & Jorge A. Pérez (2020): *Session Type Systems based on Linear Logic: Classical versus Intuitionistic*. In Stephanie Balzer & Luca Padovani, editors: *PLACES@ETAPS 2020*, *EPTCS* 314, pp. 1–11, doi:10.4204/EPTCS.314.1.

[36] Carl Hewitt, Peter Boehler Bishop, Irene Greif, Brian Cantwell Smith, Todd Matson & Richard Steiger (1973): *Actor Induction and Meta-Evaluation*. In Patrick C. Fischer & Jeffrey D. Ullman, editors: *POPL*, ACM Press, pp. 153–168, doi:10.1145/512927.512942.

[37] Daniel Hirschkoff, Enguerrand Prebet & Davide Sangiorgi (2020): *On the Representation of References in the Pi-Calculus*. In Igor Konnov & Laura Kovács, editors: *CONCUR*, *LIPIcs* 2017, Schloss Dagstuhl, pp. 34:1–34:20, doi:10.4230/LIPIcs.CONCUR.2020.34.

[38] Kohei Honda & Nobuko Yoshida (1995): *On Reduction-Based Process Semantics*. *Theor. Comput. Sci.* 151(2), pp. 437–486, doi:10.1016/0304-3975(95)00074-7.

[39] Eiichi Horita & Ken Mano (1997): *A Metric Semantics for the π-Calculus Extended with External Events*. *Kôkyûroku* 996, pp. 67–81. Available at `http://hdl.handle.net/2433/61239`.

[40] Ross Horne, Ki Yung Ahn, Shang-Wei Lin & Alwen Tiu (2018): *Quasi-Open Bisimilarity with Mismatch is Intuitionistic*. In Anuj Dawar & Erich Grädel, editors: *LICS*, ACM, pp. 26–35, doi:10.1145/3209108.3209125.

[41] Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2013): *Concurrent Flexible Reversibility*. In Matthias Felleisen & Philippa Gardner, editors: *ESOP*, *LNCS* 7792, Springer, pp. 370–390, doi:10.1007/978-3-642-37036-6_21.

[42] Ivan Lanese, Doriana Medić & Claudio Antares Mezzina (2019): *Static versus dynamic reversibility in CCS*. *Acta Inform.*, doi:10.1007/s00236-019-00346-6.

[43] Ivan Lanese, Cátia Vaz & Carla Ferreira (2010): *On the Expressive Power of Primitives for Compensation Handling*. In Andrew D. Gordon, editor: *ESOP*, *LNCS* 6012, Springer, pp. 366–386, doi:10.1007/978-3-642-11957-6_20.

[44] Kim Guldstrand Larsen & Arne Skou (1991): *Bisimulation through Probabilistic Testing*. *Inf. Comput.* 94(1), pp. 1–28, doi:10.1016/0890-5401(91)90030-6.

[45] Jean-Marie Madiot (2015): *Higher-order languages: dualities and bisimulation enhancements*. Ph.D. thesis, École Normale Supérieure de Lyon, Università di Bologna. Available at `https://hal.archives-ouvertes.fr/tel-01141067`.

[46] Massimo Merro (1998): *On the Expressiveness of Chi, Update, and Fusion calculi*. In Ilaria Castellani & Catuscia Palamidessi, editors: *EXPRESS*, *Electron. Notes Theor. Comput. Sci.* 16, Elsevier, pp. 133–144, doi:10.1016/S1571-0661(04)00122-7.

[47] Massimo Merro & Francesco Zappa Nardelli (2005): *Behavioral theory for mobile ambients*. *J. ACM* 52(6), pp. 961–1023, doi:10.1145/1101821.1101825.

[48] Robin Milner (1977): *Fully abstract models of typed λ-calculi*. *Theor. Comput. Sci.* 4(1), pp. 1–22, doi:10.1016/0304-3975(77)90053-6.

[49] Robin Milner (1980): *A Calculus of Communicating Systems*. LNCS, Springer-Verlag, doi:10.1007/3-540-10235-3.

[50] Robin Milner (1981): *A Modal Characterisation of Observable Machine-Behaviour*. In Egidio Astesiano & Corrado Böhm, editors: *CAAP '81, Trees in Algebra and Programming, 6th Colloquium, Genoa, Italy, March 5-7, 1981, Proceedings*, *LNCS* 112, Springer, pp. 25–34, doi:10.1007/3-540-10828-9_52.

[51] Robin Milner (1986): *A Calculus of Communicating Systems*. LFCS Report Series ECS-LFCS-86-7, The University of Edinburgh. Available at `http://www.lfcs.inf.ed.ac.uk/reports/86/ECS-LFCS-86-7/`.

[52] Robin Milner (1989): *Communication and Concurrency*. PHI Series in computer science, Prentice-Hall.

[53] Robin Milner (1993): *Elements of Interaction: Turing Award Lecture*. *Commun. ACM* 36(1), p. 78–89, doi:10.1145/151233.151240.

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

[55] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, II*. *Inf. Comput.* 100(1), pp. 41–77, doi:10.1016/0890-5401(92)90009-5.

[56] Robin Milner & Davide Sangiorgi (1992): *Barbed Bisimulation*. In Werner Kuich, editor: *ICALP*, *LNCS* 623, Springer, pp. 685–695, doi:10.1007/3-540-55719-9_114.

[57] Ugo Montanari & Vladimiro Sassone (1992): *Dynamic congruence vs. progressing bisimulation for CCS*. *Fund. Inform.* 16(1), pp. 171–199. Available at `https://eprints.soton.ac.uk/261817/`.

[58] Mogens Nielsen & Christian Clausen (1994): *Bisimulation for Models in Concurrency*. In Bengt Jonsson & Joachim Parrow, editors: *CONCUR '94*, *LNCS* 836, Springer, pp. 385–400, doi:10.1007/BFb0015021.

[59] Mogens Nielsen, Uffe Engberg & Kim S. Larsen (1989): *Fully abstract models for a process language with refinement*. In J. W. de Bakker, Willem P. de Roever & Grzegorz Rozenberg, editors: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, *LNCS* 354, Springer, pp. 523–548, doi:10.1007/BFb0013034.

[60] Patrick Niemeyer & Daniel Leuck (2013): *Learning Java*, 4th edition. O'Reilly Media, Incorporated.

[61] Catuscia Palamidessi & Frank D. Valencia (2005): *Recursion vs Replication in Process Calculi: Expressiveness*. *Bull. EATCS* 87, pp. 105–125. Available at `http://eatcs.org/images/bulletin/beatcs87.pdf`.

[62] Joachim Parrow (2001): *An Introduction to the π-Calculus*. In Jan A. Bergstra, Alban Ponse & Scott A. Smolka, editors: *Handbook of Process Algebra*, North-Holland / Elsevier, pp. 479–543, doi:10.1016/b978-044482830-9/50026-6.

[63] Joachim Parrow & Davide Sangiorgi (1993): *Algebraic Theories for Name-Passing Calculi*. In J. W. de Bakker, Willem P. de Roever & Grzegorz Rozenberg, editors: *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium, Noordwijkerhout, The Netherlands, June 1-4, 1993, Proceedings*, *LNCS* 803, Springer, pp. 509–529, doi:10.1007/3-540-58043-3_27.

[64] Iain Phillips & Irek Ulidowski (2007): *Reversibility and Models for Concurrency*. *Electron. Notes Theor. Comput. Sci.* 192(1), pp. 93–108, doi:10.1016/j.entcs.2007.08.018.

[65] Benjamin C. Pierce & Davide Sangiorgi (1996): *Typing and Subtyping for Mobile Processes*. *MSCS* 6(5), pp. 409–453, doi:10.1017/S096012950007002X.

[66] Davide Sangiorgi (1996): *A Theory of Bisimulation for the pi-Calculus*. *Acta Inform.* 33(1), pp. 69–97, doi:10.1007/s002360050036.

[67] Davide Sangiorgi (1999): *The Name Discipline of Uniform Receptiveness*. *Theor. Comput. Sci.* 221(1-2), pp. 457–493, doi:10.1016/S0304-3975(99)00040-7.

[68] Davide Sangiorgi (2011): *Introduction to Bisimulation and Coinduction*. CUP.

[69] Davide Sangiorgi (2011): *Pi-Calculus*. In David A. Padua, editor: *Encyclopedia of Parallel Computing*, Springer, pp. 1554–1562, doi:10.1007/978-0-387-09766-4_202.

[70] Davide Sangiorgi & Jan Rutten, editors (2011): *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9780511792588.

[71] Davide Sangiorgi & David Walker (2001): *On Barbed Equivalences in pi-Calculus*. In Kim Guldstrand Larsen & Mogens Nielsen, editors: *CONCUR*, *LNCS* 2154, Springer, pp. 292–304, doi:10.1007/3-540-44685-0_20.

[72] Davide Sangiorgi & David Walker (2001): *The Pi-calculus*. CUP.

[73] Vladimiro Sassone, Mogens Nielsen & Glynn Winskel (1996): *Models for Concurrency: Towards a Classification*. *Theor. Comput. Sci.* 170(1-2), pp. 297–348, doi:10.1016/S0304-3975(96)80710-9.

[74] Peter Selinger & Benoît Valiron (2009): *Quantum Lambda Calculus*. In Simon Gay & Ian Mackie, editors: *Semantic Techniques in Quantum Computation*, Cambridge University Press, p. 135–172, doi:10.1017/CBO9781139193313.005.

[75] Colin Stirling (1995): *Modal and Temporal Logics for Processes*. In Faron Moller & Graham M. Birtwistle, editors: *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)*, *LNCS* 1043, Springer, pp. 149–237, doi:10.1007/3-540-60915-6_5.

[76] Paul Taylor: *Comment to "Substitution is pullback"*. Available at `http://math.andrej.com/2012/09/28/substitution-is-pullback/`.

[77] André van Tondervan (2004): *A Lambda Calculus for Quantum Computation*. *SIAM J. Comput.* 33(5), pp. 1109–1135, doi:10.1137/S0097539703432165.

[78] Carlos A. Varela (2013): *Programming Distributed Computing Systems: A Foundational Approach*. The MIT Press.

[79] Glynn Winskel (2017): *Event Structures, Stable Families and Concurrent Games*. Lecture notes, University of Cambridge. Available at `https://www.cl.cam.ac.uk/~gw104/ecsym-notes.pdf`.

[80] Wang Yi (1991): *CCS + Time = An Interleaving Model for Real Time Systems*. In Javier Leach Albert, Burkhard Monien & Mario Rodríguez-Artalejo, editors: *ICALP*, *LNCS* 510, Springer, pp. 217–228, doi:10.1007/3-540-54233-7_136.

Towards Generalised Half-Duplex Systems*

Cinzia Di Giusto Loïc Germerie Guizouarn Etienne Lozes
Université Nice Côte d'Azur
CNRS, I3S
Sophia Antipolis, France
{cinzia.di-giusto,loic.germerie-guizouarn,etienne.lozes}@univ-cotedazur.fr

FIFO automata are finite state machines communicating through FIFO queues. They can be used for instance to model distributed protocols. Due to the unboundedness of the FIFO queues, several verification problems are undecidable for these systems. In order to model-check such systems, one may look for decidable subclasses of FIFO systems. Binary half-duplex systems are systems of two FIFO automata exchanging over a half-duplex channel. They were studied by Cece and Finkel who established the decidability in polynomial time of several properties. These authors also identified some problems in generalising half-duplex systems to multi-party communications. We introduce greedy systems, as a candidate to generalise binary half-duplex systems. We show that greedy systems retain the same good properties as binary half-duplex systems, and that, in the setting of mailbox communications, greedy systems are quite closely related to a multiparty generalisation of half-duplex systems.

1 Introduction

FIFO automata, also known as asynchronous communicating automata (i.e., finite state automata that exchange messages via FIFO queues) are an interesting formalism for modeling distributed protocols. In their most general formulation, these automata are Turing powerful, and in order to be able to model check them it is necessary to reduce their expressiveness.

Binary half-duplex systems, introduced by Cece and Finkel [5], are systems with two participants and a bidirectional channel formed of two FIFO queues, such that communication happens only in one direction at a time. The stereotypical half-duplex device is the walkie-talkie (or the CB). In several applications, in particular when FIFO buffers are bounded and sends may be blocking, half-duplex communications are considered a good practice to avoid send-send deadlocks. Language support for enforcing this discipline of communication includes, for instance, binary session types [14, 15] or Sing# channel contracts [10, 20].

In [5], Cece and Finkel show that (1) whether a system is half-duplex is decidable in polynomial time, (2) the set of reachable configurations is regular, and (3) properties like progress and boundedness are decidable in polynomial time. Cece and Finkel also present two possible notions 'multiparty half-duplex' systems generalizing their class to systems of any number of machines for p2p communications (one FIFO queue per pair of machine).

The first generalisation involves assuming that at most one queue over all queues is non-empty at any time. This generalisation preserves decidability but is very restrictive. The second generalisation restricts the communications between each pair of participants to half-duplex communications, that is only one buffer per bidirectional channel can be used simultaneously. This generalisation however does

*This work has been supported by the French government, through the EUR DS4H Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-17-EURE- 0004.

Submitted to:
ICE 2021

not preserve decidability: the systems captured by this definition form a Turing powerful class. In fact, with just three machines it is possible to mimic the tape of a Turing machine.

It could be believed that these results end the discussion about multi-party half-duplex systems. In this work, we claim conversely that there is another natural and relevant notion of multi-party half-duplex communications that allows us to generalise the results of Cece and Finkel. We introduce *greedy systems*, which are systems for which all executions can be rescheduled in such a way that all receptions are immediately preceded by their corresponding send. This notion is quite natural, and closely related to other notions like synchronisability [3], 1-synchronous systems [4], or existentially 1-bounded system [12] (see Section 6 for a detailed discussion).

In this work, we establish the following results:

1. whether a system is greedy is decidable in polynomial time (when the number of processes is fixed);
2. for greedy systems, all regular safety properties, which includes reachability, absence of unspecified receptions, progress, and boundedness are decidable in polynomial time.
3. we generalize binary half-duplex systems to multiparty mailbox half-duplex systems and we show that (1) mailbox half-duplex systems are greedy, and (2) greedy systems without orphan messages, at least in the binary case, are half-duplex.

The first result follows from techniques developed by Bouajjani *et al* [4] for k-synchronous systems. The main challenge here is that we address a more general model of communicating systems that encompasses both mailbox and p2p communications, but also allows any form of sharing of buffers among processes. The second result is based on an approach that, to the best of our knowledge, is new, although it borrows from some general principles from regular model-checking. The challenge is that, unlike for binary half-duplex systems, the reachability set of greedy systems is not regular, which complicates how automata-based techniques can be used to solve regular safety. The third contribution aims at answering, although incompletely, the question we would like to address with this work: what is a relevant notion of multi-party half-duplex systems?

Outline The paper is organised as follows: Section 2 introduces communicating automata and systems. Section 3 defines greedy systems and establishes the decidability of the greediness of a system. Section 4 discusses regular safety for greedy systems. Section 5 compares greedy systems and half-duplex systems, first in the binary setting, then in the multi-party setting, by introducing the notion of mailbox half-duplex systems. Finally, Section 6 concludes with some final remarks and discusses related works.

2 Preliminaries

For a finite set S, S^* denotes the set of finite words over S, $w_1 \cdot w_2$ denotes the concatenation of two words, $|w|$ denotes the length of w, and ε denotes the empty word. We assume some familiarity with non-deterministic finite state automata, and we write $\mathscr{L}(\mathscr{A})$ for the language accepted by the automaton $\mathscr{A}$. For two sets S and I, we write $\mathbf{b}$ (in bold) for an element of S^I, and b_i for the i-th component of $\mathbf{b}$, so that $\mathbf{b} = (b_i)_{i \in I}$.

A *FIFO automaton* is basically a finite state machine equipped with FIFO queues where transitions are labelled with either queuing or dequeuing actions. More precisely:

Definition 1 (FIFO automaton)**.** *A* FIFO automaton *is a tuple*[1] $\mathscr{A} = (L, \mathbb{V}, I, \mathsf{Act}, \delta, l_0)$ *where (1) L is a finite set of* control states*, (2) $\mathbb{V}$ is a finite set of* messages*, (3) I is a finite set of* buffer identifiers*, (4)* $\mathsf{Act} \subseteq I \times \{!, ?\} \times \mathbb{V}$ *is a finite set of* actions*, (5) $\delta \subseteq L \times \mathsf{Act} \times L$ is the* transition relation*, and (6) $l_0 \in L$ is the* initial control state*. The* size $|\mathscr{A}|$ *of $\mathscr{A}$ is* $|L| + |\delta|$.

Actions $a = (i, \dagger, v)$, for $\dagger \in \{!, ?\}$ are also denoted by $i \dagger v$. Given a FIFO automaton $\mathscr{A} = (L, \mathbb{V}, I, \delta, l_0)$, a *configuration* of $\mathscr{A}$ is a tuple $\gamma = (l, \mathbf{b}) \in L \times \mathbb{V}^I$. The initial configuration is $\gamma_0 = (l_0, \mathbf{b}^{\emptyset})$ where for all $i \in I$, $b_i^{\emptyset} = \varepsilon$. A *step* is a tuple (γ, a, γ') (often written $\gamma \xrightarrow[\mathscr{A}]{a} \gamma'$) where $\gamma = (l, \mathbf{b})$ and $\gamma' = (l', \mathbf{b}')$ are configurations and a is an action, such that the following holds:

- $(l, a, l') \in \delta$,
- if $a = i!v$, then $b_i' = b_i \cdot v$ and $b_j' = b_j$ for all $j \in I \setminus \{i\}$.
- if $a = i?v$, then $b_i = v \cdot b_i'$ and $b_j' = b_j$ for all $j \in I \setminus \{i\}$.

Next, we define systems of FIFO automata. We pick a very general definition, where each FIFO queue might be queued (resp. dequeued) by more than one automaton, and where an automaton might 'send a message to itself'. Most of the theory can be done in this general setting without extra cost, but we merely have in mind either mailbox systems or p2p systems (defined below).

A FIFO system, later called simply a *system*, is a family $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ of FIFO automata such that all FIFO automata have disjoint sets of actions: for all $p \neq q \in \mathbb{P}$, $\mathsf{Act}_p \cap \mathsf{Act}_q = \emptyset$. Each $p \in \mathbb{P}$ is referred to as a *process*. The condition on the disjointness of the sets of actions helps to identify the process that is responsible for a given action: for an action a, we write $\mathsf{process}(a)$ to denote the unique p (when it exists) such that $a \in \mathsf{Act}_p$.

Let us now define mailbox and p2p systems. Informally, a p2p system is a system in which each pair of processes has a dedicated buffer for exchanging messages. Instead, for mailbox communication, each process receives messages from all other processes in a single buffer. Let us now formally define these notions. To this aim, it will be useful to identify what are the buffers an automaton queues in (resp. dequeues from). Hence, for a given FIFO automaton $\mathscr{A}_p = (L_p, \mathbb{V}_p, I_p, \mathsf{Act}_p, \delta_p, l_{0,p})$, and for $\dagger \in \{!, ?\}$, we write $I_p^{\dagger}$ for the set of buffer identifiers i such that there exists $v \in \mathbb{V}_p$ such that $i \dagger v \in \mathsf{Act}_p$. A system $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ is *p2p* if for all $p \in \mathbb{P}$, $I_p \subseteq \mathbb{P}^2$, $I_p^! = \{p\} \times (\mathbb{P} \setminus \{p\})$, and $I_p^? = (\mathbb{P} \setminus \{p\}) \times \{p\}$. A system $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ is *a mailbox system* if for all $p \in \mathbb{P}$, $I_p \subseteq \mathbb{P}$, $I_p^! = \mathbb{P} \setminus \{p\}$ and $I_p^? = \{p\}$. Thus in a p2p system with n processes, there are at most $n(n-1)$ buffers, and in a mailbox system with n processes there are at most n buffers. A *binary system* is a system $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ such that $\mathbb{P} = \{1, 2\}$ and for all $p \in \mathbb{P}$, $I_p^! = \{3 - p\} = I_{3-p}^?$; note that a binary system is both p2p and mailbox. We sometimes use a more handy notation for actions of a mailbox (resp. p2p) system: if $\mathsf{process}(i!v) = p$ and $\mathsf{process}(i?v) = q$, we sometimes write $!v^{p \to q}$ instead of $i!v$ and $?v^{p \to q}$ instead of $i?v$.

Example 1 (FIFO Automata). Figure 1 shows a graphical representation of a FIFO system, borrowed from [2]. This system represents a protocol between a client, a server and a database logging requests from the client and the server. In this protocol, a client can log something on the database or send requests to the server, when those requests are satisfied the server logs them in a database. Each automaton is equipped with a buffer in which it receives messages from all other participants: this system is an example of a mailbox system. To improve readability of the graphical representation, we refer to the buffers with the initial of the automaton to which they are associated. For example, s is the buffer into which the server can receive messages. This simple system will be used as a running example throughout the paper. □

[1] Note that FIFO automata do not have accepting states, therefore they are not a special case of non-deterministic finite state automaton, and there is not such a thing as "the language of a FIFO automaton".

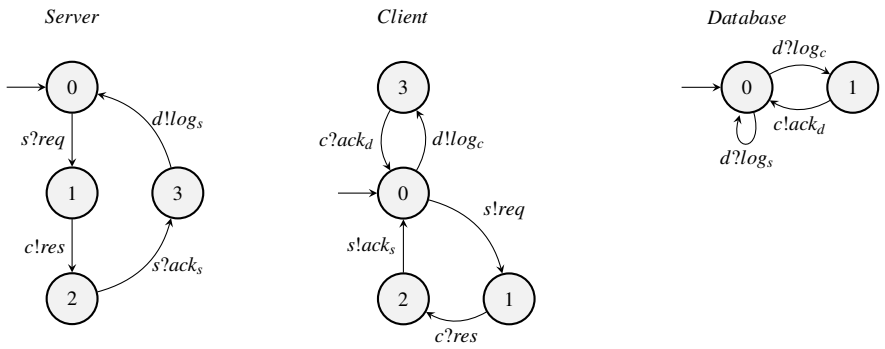


Figure 1: Client/Server/Database protocol

The size $|\mathfrak{S}|$ of $\mathfrak{S}$ is the sum of the $|\mathscr{A}_p|$. Note that $|\mathbb{P}|$ and $|I|$ are independent from the size of $\mathfrak{S}$. In particular, when we say that an algorithm is in polynomial time, we mean in time $\mathscr{O}(|\mathfrak{S}|^k)$ for some k that may depend on $|\mathbb{P}|$ and $|I|$.

The FIFO automaton $\mathsf{product}(\mathfrak{S})$ associated with $\mathfrak{S}$ is the standard asynchronous product automaton: $\mathsf{product}(\mathfrak{S}) = (\Pi_{p\in\mathbb{P}} L_p, \bigcup_{p\in\mathbb{P}} \mathbb{V}_p, \bigcup_{p\in\mathbb{P}} I_p, \delta, \mathbf{l}_0)$ where $\mathbf{l}_0 = (l_{0,p})_{p\in\mathbb{P}}$ and δ is the set of triples $(\mathbf{l}, a, \mathbf{l}')$ for which there exists $p \in \mathbb{P}$ such that $(l_p, a, l'_p) \in \delta_p$ and $l_q = l'_q$ for all $q \in \mathbb{P} \setminus \{p\}$. We often identify $\mathfrak{S}$ and $\mathsf{product}(\mathfrak{S})$ and write for instance $\xrightarrow[\mathfrak{S}]{a}$ instead of $\xrightarrow[\mathsf{product}(\mathfrak{S})]{a}$. Similarly, we say that $\gamma = (\mathbf{l}, \mathbf{b})$ is a configuration of $\mathfrak{S}$ while we mean that γ is a configuration of $\mathsf{product}(\mathfrak{S})$. An execution $e = a_1 \cdots a_n \in \mathsf{Act}^*$ is a sequence of actions. As usual $\overset{e}{\Rightarrow}$ stands for $\xrightarrow{a_1} \cdots \xrightarrow{a_n}$. We write $\mathsf{executions}(\mathfrak{S})$ for $\{e \in \mathsf{Act}^* \mid \gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma \text{ for some } \gamma\}$.

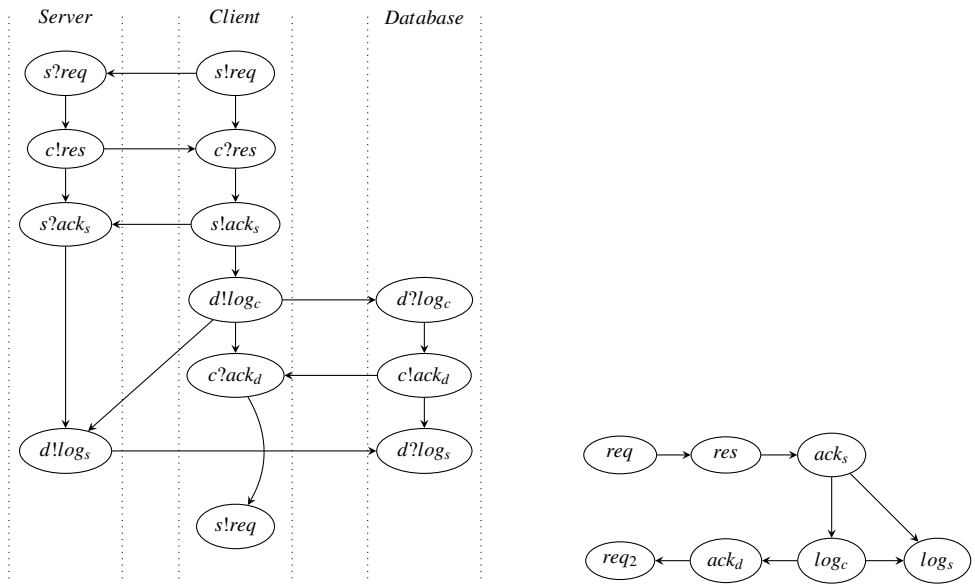
Next, we introduce the definition of reachable configuration:

Definition 2 (Reachable configuration)**.** *Let $\mathfrak{S}$ be a system. A configuration γ is reachable if there exists $e \in \mathsf{Act}^*$ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$. The set of all reachable configurations of $\mathfrak{S}$ is denoted $RS(\mathfrak{S})$.*

Given an execution $e = a_1 \cdots a_n$, we say that $\{j, j'\} \subseteq \{1, \cdots, n\}^2$ is a *matching pair* if there exists a buffer identifier i, a message v and natural number k such that (1) $a_j = i!v$, (2) $a_{j'} = i?v$, (3) a_j is the k-th send action on i in e, and (4) $a_{j'}$ is the k-th receive action on i in e. A *communication* of e is either a matching pair $\{j, j'\}$, or a singleton $\{j\}$ such that j does not belong to any matching pair (such a communication is called unmatched). We write $\mathsf{com}(e)$ to denote the set of communications of e.

An execution imposes a total order on the actions. Sometimes, however, it is useful to visualise only the causal dependencies between actions. Message sequence charts [17] are usually used to this aim, as they only depict an order between matched pairs of actions and between actions of the same process. However, message sequence charts do not represent graphically the causal dependencies due to shared buffers, like the ones found in mailbox systems. Here we define *action graphs* that depict all causal dependencies. When considering p2p communications, action graphs and message sequence charts coincide. We say that two actions a_1, a_2 commute if $\mathsf{process}(a_1) \neq \mathsf{process}(a_2)$ and it is not the case that a_1 and a_2 are two actions of the same type on a same buffer: there is no $\dagger \in \{!, ?\}$, $i \in I$ and $v_1, v_2 \in \mathbb{V}$ such that $a_1 = i\dagger v_1$ and $a_2 = i\dagger v_2$.

Definition 3 (Action graph)**.** *Given an execution $e = a_1 \cdots a_n$, the* action graph $\mathsf{agraph}(e)$ *is the vertex-labeled directed graph $(\{1, \ldots, n\}, \prec_e, \lambda_e)$ where $\lambda_e(j) = a_j$ and $j \prec_e j'$ if (1) $j < j'$ and (2) either $a_j, a_{j'}$ do not commute, or $\{j, j'\}$ is a matching pair.*



(a) The action graph $\mathsf{agraph}(e)$ (b) The conflict graph $\mathsf{cgraph}(e)$

Figure 2: Causal dependencies of an execution of Client/Server/Database protocol

When $j \prec_e j'$, we sometimes say that a_j *happens before* $a_{j'}$. Note that for an execution $e \in \mathsf{executions}(\mathfrak{S})$, $\prec_e^*$ is a partial order. Two executions e, e' are causally equivalent, denoted by $e \overset{\mathfrak{S}}{\sim} e'$, if their action graphs are isomorphic. Say differently, e, e' are causally equivalent if they are two linearisations of a same *happens before* partial order. Note that if $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$ and $e \overset{\mathfrak{S}}{\sim} e'$, then $\gamma_0 \overset{e'}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$.

Another graphical tool that we will use to talk about equivalent executions is the *conflict graph*, which is intuitively obtained from the action graph by merging matching pairs of vertices.

Definition 4 (Conflict graph)**.** *Given an execution $e = a_1 \cdots a_n$, the conflict graph $\mathsf{cgraph}(e)$ of the execution e is the directed graph $(\mathsf{com}(e), \rightarrow_e)$ where for all communications $c_1, c_2 \in \mathsf{com}(e)$, $c_1 \rightarrow_e c_2$ if there is $j_1 \in c_1$ and $j_2 \in c_2$ such that $j_1 \prec_e j_2$.*

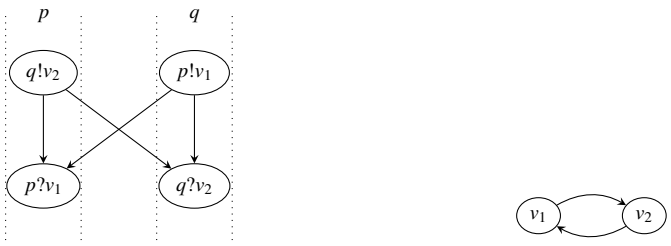
Example 2 (Action and Conflict Graphs). We go back to the system depicted in Figure 1. One of its executions is

$$e = s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot s?ack_s \cdot d!log_c \cdot d!log_s \cdot d?log_c \cdot c!ack_d \cdot c?ack_d \cdot s!req \cdot d?log_s$$

Figure 2(a) shows $\mathsf{agraph}(e)$. Actions of the same process are represented vertically between the same dotted lines. As formally explained in Definition 3, an arc from an action a and another a' means that a happens before a'. To ease readability, the arcs that follow from transitivity are omitted. For example, in a given column, there should be an arc between every pair of actions.

Figure 2(b) shows $\mathsf{cgraph}(e)$. To simplify the graph, instead of marking the matching pairs we simply identify them with the message exchanged. Message req_2 represent the second send of *req* in the execution above.

□



(a) The action graph $\mathsf{agraph}(e)$ of Example 5 (b) The conflict graph $\mathsf{cgraph}(e)$

Figure 3: Visual representations of a non-greedy execution e

3 Greedy systems

In this section we introduce greedy systems. Those systems aim at mimicking rendez-vous or synchronous communications by checking whether each execution can be rescheduled to an equivalent one where all receptions immediately follow their corresponding send.

Definition 5 (Greedy system)**.** *An execution e is* greedy *if all matching pairs are of the form $\{j, j+1\}$. A system $\mathfrak{S}$ is* greedy *if for all execution $e \in \mathsf{executions}(\mathfrak{S})$, there exists a greedy execution e' such that $e \overset{\mathfrak{S}}{\sim} e'$.*

Example 3. The execution $s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot d!log_c \cdot d?log_c$ is greedy, but the execution $s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot d!log_c \cdot s?ack_s$ is not greedy, although causally equivalent to a greedy execution. □

Example 4 (Greedy system). The system in Figure 1 is greedy. Take for instance execution e of the example 2. This execution is not greedy as messages log_c, log_s, and ack_d are not received right after their send. Still since those action can commute and by observing that the conflict graph in Figure 2(b) does not present any cycle (see Lemma 6) below), there exists an equivalent greedy execution e':

$$e' = s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot s?ack_s \cdot d!log_c \cdot d?log_c \cdot c!ack_d \cdot c?ack_d \cdot s!req \cdot d!log_s \cdot d?log_s.$$

□

Example 5. Consider a system with two processes p and q each sending a message to the other, and whose corresponding receptions only happen after the send (see Figure 3). Then the execution $e = p!v_1 \cdot q!v_2 \cdot p?v_1 \cdot q?v_2$ is not causally equivalent to a greedy execution, therefore the whole system is not greedy. □

In the remainder of this section, we show that deciding whether a system is greedy is feasible in polynomial time. The proof is in three steps: first, we give a graphical characterisation of the executions that are causally equivalent to greedy executions; second, we show that the non-greediness of a system is revealed by the existence of 'bad' executions of a certain shape, called borderline executions. Finally, we show that the graphical characterisation can be exploited to show the regularity of the language of borderline violations, from which we get the decidability of greediness.

Lemma 6. *e is causally equivalent to a greedy execution if and only if $\mathsf{cgraph}(e)$ is acyclic.*

Proof. The left to right implication follows from two observations: first, two causally equivalent executions have isomorphic conflict graphs (because they have isomorphic action graphs), and second, the conflict graph of a greedy execution is acyclic, because for a greedy execution $c_1 \rightarrow_e c_2$ induces

$\min c_1 < \min c_2$. Conversely, let $e = a_1 \cdots a_n$, and assume that $\mathsf{cgraph}(e)$ is acyclic. Let $c_1 \ll \ldots \ll c_n$ be a topological order on $\mathsf{com}(e)$, $e' = c_1 \cdots c_n$ be the corresponding greedy execution, and σ be the permutation such that $e' = a_{\sigma(1)} \cdots a_{\sigma(n)}$. The claim is that $e \overset{\mathfrak{S}}{\sim} e'$. Let j, j' be two indices of e, and let us show that $j \prec_e j'$ iff $\sigma(j) \prec_{e'} \sigma(j')$. First, $\{j, j'\}$ is a matching pair of e if and only if $\{\sigma(j), \sigma(j')\}$ is a matching pair of e', which shows the equivalence in that case. Assume now that $\{j, j'\}$ is not a matching pair of e, and let c, c' be the communications of e containing j and j' respectively. Assume also that $j \prec_e j'$, we show that $\sigma(j) \prec_{e'} \sigma(j')$ (the other implication, similar, is omitted). First, $j \prec_e j'$ implies $c \to_e c'$, which entails $\sigma(c) \to_{e'} \sigma(c')$ because e and e' have the same conflict graph. Moreover, $j \prec_e j'$ implies that a_j and $a_{j'}$ do not commute, therefore either $\sigma(j) \prec_{e'} \sigma(j')$ or $\sigma(j') \prec_{e'} \sigma(j)$. By contradiction let $\sigma(j') \prec_{e'} \sigma(j)$; then $\sigma(c') \to_{e'} \sigma(c)$, contradicting the acyclicity of the conflict graph of e', which ends the proof. □

Definition 7 (Borderline violations)**.** *An execution $e \in \mathsf{executions}(\mathfrak{S})$ is a* borderline violation *if (1) e is not causally equivalent to a greedy execution, (2) $e = e_1 \cdot i?v$ for some greedy execution e_1 and receive action $i?v$.*

Example 6 (Borderline violation). An example of a borderline violation for the system whose unique maximal execution is the one of Figure 3 is the execution

$$e = !v_2^{p \to q} \cdot !v_1^{q \to p} \cdot ?v_1^{q \to p} \cdot ?v_2^{p \to q}.$$

Figure 3 shows its action and conflict graph. The action graph makes it easy to see that any execution e' equivalent to e will require both the send actions to be done before the first reception, therefore at least one reception will not follow its matching send action. □

Lemma 8. *$\mathfrak{S}$ is greedy if and only if $\mathsf{executions}(\mathfrak{S})$ contains no borderline violation.*

Proof. Obviously, if $\mathsf{executions}(\mathfrak{S})$ contains a borderline violation, $\mathfrak{S}$ is not greedy. Conversely, assume that $\mathfrak{S}$ is not greedy, and let us show that $\mathsf{executions}(\mathfrak{S})$ contains a borderline violation. Let $e \in \mathsf{executions}(\mathfrak{S})$ be an execution that is not causally equivalent to a greedy execution and of minimal length among all such executions. Then $e = e_1 \cdot a$ with e_1 causally equivalent to a greedy execution. Let e_1' be a greedy execution causally equivalent to e_1. Then $e' = e_1' \cdot a \in \mathsf{executions}(\mathfrak{S})$. Moreover, if a is a send action, then e' is greedy, contradicting the fact that e is not causally equivalent to a greedy execution. Therefore, e' is a borderline violation. □

Let $\Sigma = I \times \{!, !?\} \times \mathbb{V}$ denote the set of communications, and let $\Sigma_? = I \times \{?\} \times \mathbb{V}$ be the set of receive actions. Then a greedy execution can be represented by a word in Σ^* and a borderline violation is represented by a word in $\Sigma^* \cdot \Sigma_?$. So now, we define two non-deterministic finite state automata over $\Sigma \cup \Sigma_?$: the first one accepts all greedy executions of a system, and the second one all borderline violations. We later explain how these automata are used to decide greediness.

Lemma 9. *Let $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ of size n and $\mathbb{V} = \bigcup_{p \in \mathbb{P}} \mathbb{V}_p$, $I = \bigcup_{p \in \mathbb{P}} I_p$ be fixed. There is a non-deterministic finite state automaton $\mathscr{A}_{gr}$ computable in time $\mathscr{O}(|\mathbb{V}||I|^2 2^{|I|} n^{|\mathbb{P}|+2})$ such that $\mathscr{L}(\mathscr{A}_{gr}) = \{e \cdot i?v \in \Sigma^* \cdot \Sigma_? \mid e \cdot i?v \in \mathsf{executions}(\mathfrak{S})$ and e is greedy$\}$.*

Proof. Let $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}, I, \mathsf{Act}, \delta_{\mathfrak{S}}, \mathbf{l}_0)$ and let $\mathscr{A}_{gr} = (L_{gr}, \delta_{gr}, l_{gr,0}, F_{gr})$ be the non-deterministic finite state automaton over Σ with $L_{gr} = L_{\mathfrak{S}} \times (\{\varepsilon\} \cup I \times v) \times 2^I \cup \{l_F\}$, $l_{gr,0} = (\mathbf{l}_0, \varepsilon, \emptyset)$, $F_{gr} = \{l_F\}$, and the transitions defined as follows. First, while reading a letter $c \in \Sigma$, $\big((\mathbf{l}, x, S), c, (\mathbf{l}', x', S')\big) \in \delta_{gr}$ if

- $(\mathbf{l}, \mathbf{b}) \overset{c}{\underset{\mathfrak{S}}{\Rightarrow}} (\mathbf{l}', \mathbf{b}')$ for some $\mathbf{b}, \mathbf{b}'$ such that for all $i \in I$, $b_i \neq \emptyset$ iff $i \in S$, and $b_i' \neq \emptyset$ iff $i \in S'$, and

- one of the two following holds
  - either $x = x'$,
  - or $x = \varepsilon$ and $x' = (i, v)$ and $c = i!v$ and $i \notin S$.

Second, while reading the letter $i?v \in \Sigma_?$, $\big((\mathbf{l}, x, S), i?v, l_F\big) \in \delta_{gr}$ if $x = (i, v)$ and $(\mathbf{l}, i?v, \mathbf{l}') \in \delta_{\mathfrak{S}}$ for some $\mathbf{l}' \in L_{\mathfrak{S}}$. Then $\mathscr{L}(\mathscr{A}_{gr})$ is as announced. Moreover, each transition of $\mathscr{A}_{gr}$ can be constructed in constant time, so $\mathscr{A}_{gr}$ can be constructed in time as announced. □

Lemma 10. *There is a non-deterministic finite state automaton $\mathscr{A}_{bv}$ computable in time $\mathscr{O}(|I|^3|\mathbb{V}|^3)$ such that $\mathscr{L}(\mathscr{A}_{bv}) = \{e \in \Sigma^* \cdot \Sigma_? \mid \mathsf{cgraph}(e)$ contains a cycle$\}$.*

Proof. Let $\mathscr{A}_{bv} = (L_{bv}, \delta_{bv}, l_{bv,0}, \{l_{bv,1}\})$ be the non-deterministic finite state automaton over $\Sigma \cup \Sigma_?$ such that $L_{bv} = \{l_{bv,0}, l_{bv,1}\} \cup \Sigma_? \times \Sigma$, and for all $c, c' \in \Sigma$, for all $a \in \Sigma_?$, for all $i \in I$, $v \in \mathbb{V}$, (1) $(l_{bv,0}, c, l_{bv,0}) \in \delta_{bv}$ (2) $(l_{bv,0}, i!v, (i?v, i!v)) \in \delta_{bv}$, (3) $((a,c), c', (a,c)) \in \delta_{bv}$ (4) $((a,c), c', (a,c')) \in \delta_{bv}$ if $\mathsf{process}(c) \cap \mathsf{process}(c') \neq \emptyset$, and (5) $((i?v, c), i?v, l_{bv,1}) \in \delta_{bv}$ if $\mathsf{process}(c) \cap \mathsf{process}(i?v) \neq \emptyset$. Then $\mathscr{L}(\mathscr{A}_{bv})) = \{e \in \Sigma^*\Sigma_? \mid \mathsf{cgraph}(e)$ contains a cycle$\}$. Moreover, each transition of $\mathscr{A}_{bv}$ can be constructed in constant time, so $\mathscr{A}_{bv}$ can be constructed in time as announced. □

From the computability of the two previous automata, we deduce the decidability of system greediness.

Theorem 11. *Whether a system $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ of size n is greedy is decidable in time $\mathscr{O}(|I|^5|\mathbb{V}|^4 2^{|I|} n^{|\mathbb{P}|+2})$.*

Proof. Let $\mathscr{A}_{gr}$ and $\mathscr{A}_{bv}$ be the two automata defined in Lemmas 9 and 10. By Lemma 6 and by definition of a borderline violation, the set of borderline violations of $\mathfrak{S}$ is $\mathscr{L}(\mathscr{A}_{gr}) \cdot \Sigma_? \cap \mathscr{L}(\mathscr{A}_{bv})$. So, by Lemma 8, $\mathfrak{S}$ is greedy if and only if $\mathscr{L}(\mathscr{A}_{gr}) \cdot \Sigma_? \cap \mathscr{L}(\mathscr{A}_{bv}) = \emptyset$. The claim then directly follows from the fact that emptiness testing for a non-deterministic finite state automaton of size n is in time $\mathscr{O}(n)$. □

4 Model-Checking Greedy Systems

In this section we explore how to verify various safety properties of greedy systems in polynomial time. Since the reachability set of a greedy system is not regular, it is not obvious that regular safety properties are always decidable. We show that this problem actually is decidable, with a polynomial time complexity under mild assumptions. Then, we list a few regular safety properties that were also considered in other works, in particular for approaches based on session types [16, 8, 22].

4.1 Checking Regular Safety Properties

Let $\mathfrak{S} = (\mathscr{A}_p)_{p \in \mathbb{P}}$ with $I = \bigcup_{p \in \mathbb{P}} I_p = \{1, \ldots, |I|\}$ be a system. We identify a word $w = \mathbf{l} \cdot \sharp \cdot b_1 \cdot \sharp \cdots \sharp \cdot b_{|I|} \in L_{\mathfrak{S}} \cdot (\sharp \cdot \mathbb{V}^*)^{|I|}$ with the configuration $(\mathbf{l}, b_1, \ldots, b_{|I|})$. We say that a set of configurations $P(\mathfrak{S})$ is *regular*[2] if the corresponding set of words coding these configurations is regular. A *property* is a function P that associates to every system $\mathfrak{S}$ a set of configurations $P(\mathfrak{S})$. We say that P is regular if $P(\mathfrak{S})$ is regular for all $\mathfrak{S}$, and *computable* in time $\mathscr{O}(f(n))$ if a non-deterministic finite state automaton $\mathscr{A}$ accepting $P(\mathfrak{S})$ can be computed in time $\mathscr{O}(f(|\mathfrak{S}|))$. The *$P$ safety problem* is whether a system $\mathfrak{S}$ is such that $RS(\mathfrak{S}) \cap P(\mathfrak{S}) = \emptyset$. Examples of safety problems are discussed below in Section 4.2.

[2] also called *channel recognizable* by Cece and Finkel [5, Definition 10]

Cece and Finkel showed that, for a binary half-duplex system $\mathfrak{S}$, $RS(\mathfrak{S})$ is regular and computable in polynomial time [5, Theorem 26]. Since the emptiness of the intersection of two polynomial time computable regular languages is decidable in polynomial time, for any regular polynomial time property P, the P safety problem is decidable in polynomial time for binary half-duplex systems.

For greedy systems, however, the situation is a bit different. Indeed, observe that $RS(\mathfrak{S})$ may be non-regular and even context sensitive (it is easy to define a system with one machine and 3 buffers, such that for some control state l, $RS(\mathfrak{S}) \cap l(\sharp\mathbb{V}^*)^3 = \{la^n\sharp b^n\sharp c^n \mid n \geq 0\}$). So it is not obvious how to decide the emptiness of $RS(\mathfrak{S}) \cap P(\mathfrak{S})$.

Still, the P safety problem remains decidable for a computable regular property P.

Theorem 12. *Let P be a computable regular property. Then, for a given greedy system $\mathfrak{S} = (\mathscr{A}_p)_{p\in\mathbb{P}}$, the P safety problem is decidable. Moreover, if P is computable in time $\mathscr{O}(|\mathfrak{S}|^k)$ for some $k \geq 0$, then the problem is decidable in time $\mathscr{O}(|\mathfrak{S}|^{k+|\mathbb{P}|+2})$.*

Proof. Let $\mathfrak{S}$ be fixed, with $\mathsf{product}(\mathfrak{S}) = (L_{\mathfrak{S}}, \mathbb{V}, I, \mathsf{Act}, \delta_{\mathfrak{S}}, \mathbf{l}_{\mathfrak{S},0})$. Let $\mathscr{A} = (L_{\mathscr{A}}, \delta_{\mathscr{A}}, l_{\mathscr{A},0}, F_{\mathscr{A}})$ be the polynomial time computable non-deterministic finite state automaton over alphabet $L_{\mathfrak{S}} \cup \{\sharp\} \cup \mathbb{V}$ such that $\mathscr{L}(\mathscr{A}) = P(\mathfrak{S})$. We define an automaton $\mathscr{A}_P = (L_P, \delta_P, L_{P,0}, F_P)$ over the alphabet Σ of communications such that for all greedy execution $e \in \mathsf{executions}(\mathfrak{S})$, $e \in \mathscr{L}(\mathscr{A}_P)$ iff there is $\gamma \in P(\mathfrak{S})$ such that $\gamma_0 \overset{e}{\underset{\mathfrak{S}}{\Rightarrow}} \gamma$.

Before defining $\mathscr{A}_p$ formally, let us give some intuitions about how it works on an example. Assume that $\mathscr{A}_p$ reads $e = 1!a \cdot 2!?b \cdot 2!c \cdot 1!d$, and that the final configuration γ is $\mathbf{l}\sharp ad\sharp c$. While reading e, $\mathscr{A}_p$ should check the existence of an accepting run of $\mathscr{A}$ on γ. When $\mathscr{A}_P$ reads a communication, there are two cases. Either the communication is a matched send (like $2!?b$) and therefore it does not contribute to the final configuration, so $\mathscr{A}_p$ merely ignores it. Or the communication is an unmatched send, and it contributes to a piece of the accepting run of γ on $\mathscr{A}$. However, these pieces of the accepting run of $\mathscr{A}$ are not necessarily consecutive. For instance, in the above execution, a and d are consecutive in the run of $\mathscr{A}$, but $\mathscr{A}_p$ reads c in between, although c contributes only later to the run of $\mathscr{A}$. To correctly check the existence of a run of $\mathscr{A}$ on γ, $\mathscr{A}_p$ uses for each buffer a distinguished "pebble" placed on a state of $\mathscr{A}$. Every time $\mathscr{A}_p$ reads an unmatched send $i!v$, it moves the i-th pebble along an v transition of $\mathscr{A}$. So each pebble checks for a piece of the accepting run of the whole word coding the final configuration. $\mathscr{A}_p$ therefore also needs to make sure that all of these pieces of runs can be concatenated to form a run of $\mathscr{A}$. Therefore, at the beginning, $\mathscr{A}_p$ guesses an initial control state l_i and a final control state l'_i for each pebble, and ensures that $(l'_i, \sharp, l_{i+1}) \in \delta_{\mathscr{A}}$. While doing so, going back to our example, $\mathscr{A}_p$ ensures that $ad\#c$ can be read by $\mathscr{A}$. It remains also to deal with the control state: indeed, $\mathscr{A}$ should accept $\mathbf{l}\sharp ad\sharp c$. So $\mathscr{A}_P$ guesses before reading e that the control state will be $\mathbf{l}$ after executing e, and while reading e, it computes the current control state of $\mathfrak{S}$. In the end, it checks that this control state actually is $\mathbf{l}$.

Now that we presented the intuitions about $\mathscr{A}_p$, let us define it formally. Let us start with the set of control states. Let $L_P = L_{\mathfrak{S}} \times L_{\mathfrak{S}} \times L_{\mathscr{A}}^{|I|} \times L_{\mathscr{A}}^{|I|}$. Intuitively, the control state $(\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_F, \mathbf{l}_{\mathscr{A}}, \mathbf{l}_I)$ of $\mathscr{A}_p$ corresponds to a situation where: (1) the current control state of $\mathfrak{S}$ is $\mathbf{l}_{\mathfrak{S}}$, (2) the guessed final control state of $\mathfrak{S}$ is $\mathbf{l}_F$, (3) the i-th pebble currently is on state $l_{\mathscr{A},i}$ of $\mathscr{A}$, and (4) $\mathbf{l}_I$ is a copy of the initial positions of the pebbles and will be checked against their final positions in the end to ensure that all pieces of runs can be concatenated.

Let us now define the set $L_{P,0}$ of initial control states of $\mathscr{A}_P$. Let us set that $(\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_F, \mathbf{l}_{\mathscr{A}}, \mathbf{l}_I \in L_{P,0}$ if (1) $\mathbf{l}_{\mathscr{A}} = \mathbf{l}_I$, (2) $l_{\mathscr{A},1} \in \delta^*_{\mathscr{A}}(l_{\mathscr{A},0}, \mathbf{l}_F \cdot \sharp)$, and (3) $\mathbf{l}_{\mathfrak{S}} = \mathbf{l}_{\mathfrak{S},0}$. Intuitively, a control state is initial if (1) $\mathbf{l}_I$ is a copy of $\mathbf{l}_{\mathscr{A}}$, (2) the position of the pebble of buffer 1 is on a state that is reachable in $\mathscr{A}$ after reading $\mathbf{l}_F\sharp$ and (3) $\mathbf{l}_{\mathfrak{S}}$ is the initial control state of $\mathfrak{S}$.

Similarly, let us now define the set F_P of final control states of $\mathscr{A}_P$. Let us set $(\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_F, \mathbf{l}_{\mathscr{A}}, \mathbf{l}_I) \in F_P$ if (1) $\mathbf{l}_{\mathfrak{S}} = \mathbf{l}_F$, (2) for all $i = 1, \ldots, |I| - 1$, $(l_{\mathscr{A},i}, \sharp, l_{\mathscr{A},i+1}) \in \delta_{\mathscr{A}}$, and (3) $l_{\mathscr{A},|I|} \in F_{\mathscr{A}}$. Intuitively, a control state is final if (1) the current control state of $\mathfrak{S}$ corresponds to the guessed final one, (2) the i-th pebble can be moved along a $\sharp$ transition so as to reach the initial position of pebble $i+1$, and (3) the pebble of the last buffer reached an accepting state of $\mathscr{A}$.

Let us now define the set δ_P of transitions of $\mathscr{A}_p$. Let us set that

$$\Big((\mathbf{l}_{\mathfrak{S}}, \mathbf{l}_F, \mathbf{l}_{\mathscr{A}}, \mathbf{l}_I),\ c,\ (\mathbf{l}'_{\mathfrak{S}}, \mathbf{l}'_F, \mathbf{l}'_{\mathscr{A}}, \mathbf{l}'_I)\Big) \ \in \ \delta_P$$

if (1) $\mathbf{l}_F = \mathbf{l}'_F$, (2) $\mathbf{l}_I = \mathbf{l}'_I$, (3) $\mathbf{l}'_{\mathfrak{S}} \in \delta^*_{\mathfrak{S}}(\mathbf{l}_{\mathfrak{S}}, c)$, (4.1) if $c = i!?v$, then $\mathbf{l}_{\mathscr{A}} = \mathbf{l}'_{\mathscr{A}}$, and (4.2) if $c = i!v$, then $(l_{\mathscr{A},i}, v, l'_{\mathscr{A},i}) \in \delta_{\mathscr{A}}$ and $l_{\mathscr{A},j} = l'_{\mathscr{A},j}$ for all $j \neq i$. Intuitively, when it reads a matched send $i!?v$, $\mathscr{A}_p$ only updates $\mathbf{l}_{\mathfrak{S}}$ according to the sequence of actions $i!?v$, while when it reads an unmatched send $i!v$ it also updates the position of the i-th token.

Now that $\mathscr{A}_P$ is defined, observe that $RS(\mathfrak{S}) \cap P(\mathfrak{S}) = \emptyset$ iff $\mathscr{L}(\mathscr{A}_p) \cap \mathscr{L}(\mathscr{A}_{gr}) = \emptyset$, where $\mathscr{A}_{gr}$ is the automaton defined in Lemma 9. The emptiness of this intersection is decidable in time $\mathscr{O}(|\mathscr{A}_P| \cdot |\mathscr{A}_{gr}|)$, which shows the claim. □

4.2 Examples of Regular Safety Problems

In this section we review a few properties of systems that are polynomial-time computable regular properties and showcase some applications of Theorem 12.

Reachability The control state reachability problem is to decide, given a system $\mathfrak{S}$ and a control state $\mathbf{l} \in L_{\mathfrak{S}}$, whether there exists $\mathbf{b} \in (\mathbb{V}^*)^I$ and $e \in \mathsf{Act}^*$ such that $\gamma_0 \xRightarrow[\mathfrak{S}]{e} (\mathbf{l}, \mathbf{b})$. The configuration reachability problem, on the other hand, is to decide, given a system $\mathfrak{S}$ and a configuration γ, whether $\gamma \in RS(\mathfrak{S})$. Both problems are safety problems for a regular property P computable in polynomial time (and even constant time): $P(\mathfrak{S}) = \mathbf{l} \cdot (\sharp \cdot \mathbb{V}^*)^{|I|}$ for the control state reachability problem, and $P(\mathfrak{S}) = \{\gamma\}$ for the configuration reachability problem.

Unspecified reception Unspecified receptions is one of the errors that session types usually forbid. This error makes more sense for mailbox systems, so let us assume for now that $\mathfrak{S}$ is a mailbox system. A configuration is an *unspecified reception* if one of the participants is in a receiving state, and none of its outgoing transitions can receive the first message in its buffer.

Let us define these notions more formally. A control state l_p of process p is a receiving state if for all a, l' such that $(l_p, a, l') \in \delta_p$, a is a receive action. The set $\{v \mid (l_p, p?v, l') \in \delta_p \text{ for some } l'\}$ is called the ready set of l_p.

A configuration $(\mathbf{l}, \mathbf{b})$ is said an *unspecified reception configuration* if there is $p \in \mathbb{P}$ such that (1) l_p is a receiving state, (2) $b_p = vb'$ for some $v \in \mathbb{V}$ and $b' \in \mathbb{V}^*$, and (3) v is not in the ready set of l_p. It can be observed that the set $UR(\mathfrak{S})$ of unspecified receptions of $\mathfrak{S}$ defines a regular property that is computable in polynomial time.

Progress Another property that is central in session types is progress. A global control state $\mathbf{l}$ of $\mathfrak{S}$ is *final* if there is no action a and global control state $\mathbf{l}'$ such that $(\mathbf{l}, a, \mathbf{l}') \in \delta_{\mathfrak{S}}$. A configuration $\gamma = (\mathbf{l}, \mathbf{b})$ of $\mathfrak{S}$ *satisfies progress* if either $\mathbf{l}$ is final or there is a configuration γ' and an action a such that $\gamma \xrightarrow[\mathfrak{S}]{a} \gamma'$. A system satisfies progress if all reachable configurations satisfy progress. It can be observed that the set $NP(\mathfrak{S})$ of configurations that do not satisfy progress is regular and polynomial time computable.

4.3 Boundedness

There are other examples of properties that are regular and polynomial time, but some interesting ones are not safety properties. To conclude this section, we consider one of these properties.

Definition 13 (Boundedness)**.** *Let $\mathfrak{S}$ be a system and $k \geq 0$. A channel $i \in I$ is k-*bounded *if for all $(\mathbf{l},\mathbf{b}) \in RS(\mathfrak{S})$ $|b_i| \leq k$. $\mathfrak{S}$ is k-bounded if for all $i \in I$, i is k-bounded.*

Theorem 14. *Whether there exists $k \geq 0$ such that a greedy system $\mathfrak{S}$ is k-bounded is decidable in polynomial time. Moreover, k is computable in polynomial time.*

Proof. Let $\Sigma_! \subseteq \Sigma$ be the set of unmatched communications, and $\sigma : \Sigma^* \to \Sigma_!^*$ the morphism that erase all matched communications. By Lemma 9, and by the closure of regular languages under morphisms, there is a polynomial time computable non-deterministic finite state automaton $\mathscr{A}$ such that $\mathscr{L}(\mathscr{A}) = \{\sigma(e) \mid e \in \mathsf{executions}(\mathfrak{S}) \text{ is greedy}\}$. Then $\mathfrak{S}$ is k-bounded for some k if and only if $\mathscr{L}(\mathscr{A})$ is finite, or equivalently if and only if $\mathscr{A}$, once pruned (removing states that are not reachable from the initial state and co-reachable from a final state), is acyclic.This is decidable in time $\mathscr{O}(|\mathscr{A}|)$, and the maximal length k of a word of $\mathscr{L}(\mathscr{A})$ also is computable in time $\mathscr{O}(|\mathscr{A}|)$. □

5 Mailbox Half-Duplex Systems

Binary half-duplex systems, (called simply half-duplex by Cece and Finkel [5]) are binary systems such that all reachable configurations (l_1,l_2,b_1,b_2) are such that either $b_1 = \varepsilon$ or $b_2 = \varepsilon$. In the previous sections, we established that greedy systems enjoy the same decidability and complexity results as binary half-duplex systems. In this section, we defend the claim that greedy systems could also merit the name of multiparty half-duplex systems.

First, observe that binary half-duplex systems are greedy (see [5, Lemma 20]). The converse does not hold in general: some binary greedy systems are not half-duplex. However, under an extra hypothesis, both are equivalent. A system is called *without orphan messages* if for all reachable configuration containing a message in a buffer, it is possible to reach a configuration where this message has been received. This property is also enforced by session types and is very natural in communicating systems. Then observe that, for a given a binary system $\mathfrak{S}$ without orphan messages, the following two are equivalent: (1) $\mathfrak{S}$ is binary half-duplex, and (2) $\mathfrak{S}$ is greedy.

Let us now consider multiparty systems. Cece and Finkel proposed two notions of multiparty half-duplex systems, but conclude that they were not well behaved (one being too restrictive, and the other Turing powerful). Both of these generalisations relied on peer-to-peer communication. We propose to consider mailbox communication instead.

Definition 15 (Half-duplex execution)**.** *Fix a mailbox system $\mathfrak{S}$. An execution $e = (\mathbf{l_0},\mathbf{b}_0) \xrightarrow{a_1} (\mathbf{l_1},\mathbf{b}_1) \to \cdots \xrightarrow{a_n} (\mathbf{l_n},\mathbf{b}_n)$ is* half-duplex *if for all $j = 1,\ldots,n$, if a_j is a send action, then $b_p^{i-1} = \varepsilon$, where $p = \mathsf{process}(a_j)$.*

Intuitively, an execution is half-duplex if every process empties its queue of messages before sending.

Definition 16 (Mailbox half-duplex system)**.** *A mailbox system $\mathfrak{S}$ is mailbox half-duplex if for all execution $e \in \mathsf{executions}(\mathfrak{S})$, there is a half-duplex execution e' such that $e \overset{\mathfrak{S}}{\sim} e'$.*

Example 7. The system in Figure 1 is half-duplex. Indeed even if execution e of Example 2 is not half-duplex, by considering one of its greedy equivalents e'':

$$e'' = s!req \cdot s?req \cdot c!res \cdot c?res \cdot s!ack_s \cdot s?ack_s \cdot d!log_c \cdot d?log_c \cdot c!ack_d \cdot c?ack_d \cdot d!log_s \cdot d?log_s \cdot s!req$$

we obtain a half-duplex execution. Notice that execution e' of Example 4 is greedy but not half-duplex as the send of message log_s is done when the buffer of the Server is filled with message *req*. □

Note that a binary system is mailbox half-duplex if and only if it is binary half-duplex. In the remainder, we therefore sometimes say simply half-duplex instead of binary half-duplex or mailbox half-duplex.

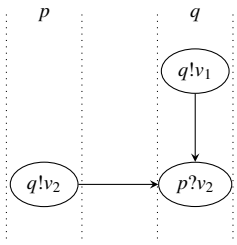
Theorem 17. *Mailbox half-duplex systems are greedy.*

Proof. We reason by contradiction. Assume that $\mathfrak{S}$ is not greedy, we show that $\mathfrak{S}$ is not half-duplex. Let $e \in \mathsf{executions}(\mathfrak{S})$ be any execution that is not causally equivalent to a greedy execution (for instance, take for e a borderline violation). We claim that for all e' such that $e \overset{\mathfrak{S}}{\sim} e'$, e' is not half-duplex. Since e' is not causally equivalent to a greedy execution, we get, by Lemma 6, that $\mathsf{cgraph}(e')$ contains a cycle of communications $c_1 \rightarrow_{e'} c_2 \rightarrow_{e'} \ldots \rightarrow_{e'} c_n \rightarrow_{e'} c_1$ where for all $i = 1, \ldots n$, either $c_i = \{j_i, k_i\}$ is a matching pair, or $c_i = \{j_i\}$ is an unmatched send. We assume that $j_i < k_i$, i.e. j_i is the index of the send action and k_i the index of the receive action. Up to a circular permutation, we can also assume, without loss of generality, that j_1 is the first send among them in e', i.e. $j_1 < j_\ell$ for all $\ell = 2, \ldots, n$. Now, let us reason by case analysis on the nature of the conflict edge $c_n \rightarrow_{e'} c_1$.

- case "$c_n \xrightarrow{SS} c_1$": $j_n \prec_{e'} j_1$. Then $j_n < j_1$, contradicts the minimality of j_1. Impossible.
- case "$c_n \xrightarrow{RS} c_1$": $k_n \prec_{e'} k_1$. Then $j_n < k_n < j_1$, impossible.
- case "$c_n \xrightarrow{RR} c_1$": $k_n \prec_{e'} k_1$. Then $k_n < k_1$ and either (1) $\mathsf{process}(a_{k_n}) = \mathsf{process}(a_{k_1})$ or (2) there is $i \in I$, $v, v' \in \mathbb{V}$ such that $a_{k_n} = i?v$ and $a_{k_1} = i?v'$. Because of the mailbox semantics, (1) and (2) are equivalent, so (2) is granted. But then $a_{j_n} = i!v$ and $a_{j_1} = i!v'$. Since e' is a FIFO execution, and $k_n < k_1$, we get that $j_n < j_1$, and again the contradiction.
- case "$c_n \xrightarrow{SR} c_1$": $j_n \prec_{e'} k_1$. Then $j_n < k_1$, and $\mathsf{process}(a_{j_n}) = \mathsf{process}(a_{k_1})$. Moreover, $j_1 < j_n$ by the minimality of j_1. To sum up, let p, q, r, v_1, v_2 be such that $a_{j_1} = !v_1^{p \to q}$, $a_{k_1} = ?v_1^{p \to q}$, and $a_{j_n} = !v_2^{q \to r}$. Then we just showed that $e' = \ldots !v_1^{p \to q} \ldots !v_2^{q \to r} \ldots ?v_1^{p \to q} \ldots$, so e' is not a half-duplex execution.

□

Notice that the converse of Theorem 17 does not hold: being greedy is not a sufficient condition to be half-duplex. Indeed, an unmatched send can fill the buffer of a process willing to send. More precisely, consider the action graph on the right. It depicts a greedy system that is not half-duplex: in fact the buffer for q is not empty when v_2 is sent. We conjecture that this is the only pathological situation, and that, like in the binary setting, if a system is greedy and has no orphan messages, then it is half-duplex.



Example 8. To finish this section we present another small example of a half-duplex system: the classic Client/Seller/Bank protocol. This system is shown in Figure 4. In this protocol a client can ask the price of an item to the seller (*ask_price* message), and receive the answer. Whenever the client agrees on a price it can place an order (via the message *buy*). Receiving this message the seller initiates a transaction with the bank. The bank asks the client for its credentials, and when it receives them it either authorizes or refuses the transaction, and notifies the seller accordingly. The seller then confirms or cancels the transaction, sending a message to the client. □

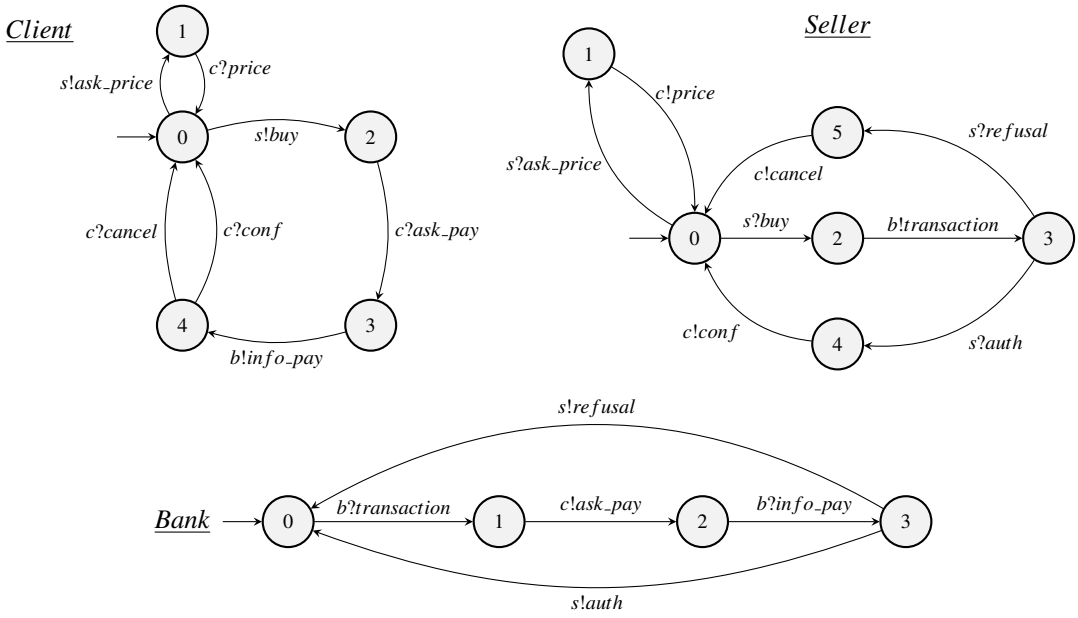


Figure 4: Client/Seller/Bank

6 Conclusion

We have introduced greedy systems, a new class of communicating systems, generalising the notion of half-duplex systems to any number of processes, and to an arbitrary model of FIFO communication (encompassing both p2p and mailbox communications). We have shown that the greediness of a system is decidable in polynomial time, and that for greedy systems regular safety properties, such as reachability, progress, and boundedness are decidable in polynomial time. Finally, we defined mailbox half-duplex systems and showed that greedy systems are intimately related to mailbox half-duplex systems.

Still, the picture is a bit incomplete: we did not address liveness properties nor more general temporal properties, and we also did not propose a notion of p2p half-duplex systems that would enjoy all desirable properties. Also, we did not report on experimental evaluation. We leave these questions for future work.

Pachl gave a general decidability result for systems of communicating finite state machines whose reachability set is regular [21]. Although the algorithm is rather brute-force, the MCSCM tool illustrates that it is amenable to an efficient CEGAR optimization [13]. Beyond regularity, and relying on visibly pushdown languages, La Torre *et al* [26] established that bounded context-switch reachability is also decidable in time exponential in the number of states and doubly exponential in the number of switch. We conjecture that bounded context-switch reachability is complete for greedy systems with a number of switch polynomial in the size of the system.

Several authors considered communicating systems where queues are not plain FIFO but may over-approximate a FIFO behaviour. A representative example of this approach is lossy channels, introduced independently in [6] and [1]. Another example is the bag semantics of buffers where messages are received without loss but out of order. Examples of uses of bag buffers can be found in [25]. Both for lossy and bag systems the reachability problem is decidable but with a high complexity: non primitive recursive for lossy [23, 24]. The exact complexity seems unknown for bag systems, but by reduction to Petri nets it is non-elementary [7].

Aside bounded context-switch model-checking, another form of bounded model-checking has been promoted by Muscholl *et al.*: the notions of universally and existentially bounded message sequence charts [19]. This leads to two notions of universally/existentially bounded systems (both having unfortunately the same name), depending whether only the MSCs leading to final stable configurations are considered [12] or all MSCs [18]. For the latter definition, reachability and membership are decidable in PSPACE. Greedy systems are existentially 1-bounded, but the converse does not hold (for instance, the system of Example 5) is existentially 1-bounded). A system is k-synchronous [4] if the message sequence charts of the system are formed of blocks of k messages. In particular, a system is 1-synchronous if the message lines never cross. For systems with p2p communications, greediness is the same as 1-synchronizability. However, for mailbox communications, some subtle examples are 1-synchronous but not greedy (see *e.g.* [9, Example 1.2]). For k-synchronous systems, reachability is decidable in PSPACE. Finally, greedy system are synchronizable in the sense of Basu and Bultan [3], but synchronizability is not decidable [11]. We believe that greediness is the notion that Basu and Bultan were aiming at with the notion of synchronizability. It might be wondered if greedy systems were not implicit in Cece and Finkel's work. Actually, some of their arguments rely on the fact that for half-duplex systems, every execution is 'reachability equivalent' to a synchronous execution. This is not exactly the notion of greedy systems we introduced, and our notion of greedy systems is closer to Bouajjani et al notion of 1-synchronous systems, although, as we just explained, they are not the same in some corner cases.

Session types are intimately related to half-duplex systems in the binary setting [20]. Several multiparty extensions of session types have been proposed, the last proposal being [22]. It seems there are also some similarities between multiparty session types and greedy systems. For instance, the notion of greedy execution shares some similarities with the notion of alternation in [8]. The study of the exact relationship between greedy systems and multi-party session typed systems is left for future work.

We would like to thank all the ICE reviewers for their comments that greatly improved the present paper.

References

[1] Parosh Aziz Abdulla & Bengt Jonsson (1996): *Verifying Programs with Unreliable Channels*. *Inf. Comput.* 127(2), pp. 91–101. Available at `https://doi.org/10.1006/inco.1996.0053`.

[2] Lakhdar Akroun & Gwen Salaün (2018): *Automated verification of automata communicating via FIFO and bag buffers*. *Formal Methods Syst. Des.* 52(3), pp. 260–276. Available at `https://doi.org/10.1007/s10703-017-0285-8`.

[3] Samik Basu & Tevfik Bultan (2016): *On deciding synchronizability for asynchronously communicating systems*. *Theor. Comput. Sci.* 656, pp. 60–75, doi:10.1016/j.tcs.2016.09.023. Available at `https://doi.org/10.1016/j.tcs.2016.09.023`.

[4] Ahmed Bouajjani, Constantin Enea, Kailiang Ji & Shaz Qadeer (2018): *On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, *Lecture Notes in Computer Science* 10982, Springer, pp. 372–391, doi:10.1007/978-3-319-96142-2_23. Available at `https://doi.org/10.1007/978-3-319-96142-2_23`.

[5] Gérard Cécé & Alain Finkel (2005): *Verification of programs with half-duplex communication*. *Inf. Comput.* 202(2), pp. 166–190. Available at `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/CF-icomp05.pdf`.

[6] Gérard Cécé, Alain Finkel & S. Purushothaman Iyer (1996): *Unreliable Channels are Easier to Verify Than Perfect Channels*. *Inf. Comput.* 124(1), pp. 20–31, doi:10.1006/inco.1996.0003. Available at https://doi.org/10.1006/inco.1996.0003.

[7] Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux & Filip Mazowiecki (2021): *The Reachability Problem for Petri Nets Is Not Elementary*. *J. ACM* 68(1), pp. 7:1–7:28, doi:10.1145/3422822. Available at https://doi.org/10.1145/3422822.

[8] Pierre-Malo Deniélou & Nobuko Yoshida (2012): *Multiparty Session Types Meet Communicating Automata*. In Helmut Seidl, editor: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *Lecture Notes in Computer Science* 7211, Springer, pp. 194–213, doi:10.1007/978-3-642-28869-2_10. Available at https://doi.org/10.1007/978-3-642-28869-2_10.

[9] Cinzia Di Giusto, Laetitia Laversa & Étienne Lozes (2020): *On the k-synchronizability of Systems*. In: *FOSSACS 2020*, *LNCS* 12077, pp. 157–176. Available at https://arxiv.org/abs/1909.01627.

[10] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus & Steven Levi (2006): *Language support for fast and reliable message-based communication in singularity OS*. In Yolande Berbers & Willy Zwaenepoel, editors: *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, ACM, pp. 177–190, doi:10.1145/1217935.1217953. Available at https://doi.org/10.1145/1217935.1217953.

[11] Alain Finkel & Étienne Lozes (2017): *Synchronizability of Communicating Finite State Machines is not Decidable*. In Ioannis Chatzigiannakis, Piotr Indyk, Anca Muscholl & Fabian Kuhn, editors: *Proceedings of the 44th International Colloquium on Automata, Languages and Programming (ICALP'17)*, *Leibniz International Proceedings in Informatics* 80, Leibniz-Zentrum für Informatik, Warsaw, Poland, pp. 122:1–122:14, doi:10.4230/LIPIcs.ICALP.2017.122. Available at http://drops.dagstuhl.de/opus/volltexte/2017/7402.

[12] Blaise Genest, Dietrich Kuske & Anca Muscholl (2007): *On Communicating Automata with Bounded Channels*. *Fundam. Inform.* 80(1-3), pp. 147–167. Available at http://perso.crans.org/~genest/GKM07.pdf.

[13] Alexander Heußner, Tristan Le Gall & Grégoire Sutre (2012): *McScM: A General Framework for the Verification of Communicating Machines*. In Cormac Flanagan & Barbara König, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *Lecture Notes in Computer Science* 7214, Springer, pp. 478–484, doi:10.1007/978-3-642-28756-5_34. Available at https://doi.org/10.1007/978-3-642-28756-5_34.

[14] Kohei Honda (1993): *Types for Dyadic Interaction*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35. Available at https://doi.org/10.1007/3-540-57208-2_35.

[15] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567. Available at https://doi.org/10.1007/BFb0053567.

[16] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695. Available at https://doi.org/10.1145/2827695.

[17] *ITU-TS: ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS (1999).*

[18] D. Kuske & A. Muscholl (2019): *Communicating Automata*. In J.-E. Pin, editor: *Handbook of Automata: from Mathematics to Applications (AutoMathA)*, European Science Foundation. Available at https://eiche.theoinf.tu-ilmenau.de/kuske/Submitted/cfm-final.pdf.

[19] Markus Lohrey & Anca Muscholl (2004): *Bounded MSC communication*. *Inf. Comput.* 189(2), pp. 160–181, doi:10.1016/j.ic.2003.10.002. Available at https://doi.org/10.1016/j.ic.2003.10.002.

[20] Étienne Lozes & Jules Villard (2011): *Reliable Contracts for Unreliable Half-Duplex Communications*. In: *WS-FM 2011*, *LNCS* 7176, pp. 2–16. Available at https://doi.org/10.1007/978-3-642-29834-9.

[21] Jan K. Pachl (2003): *Reachability problems for communicating finite state machines*. *CoRR* cs.LO/0306121. Available at http://arxiv.org/abs/cs/0306121.

[22] Alceste Scalas & Nobuko Yoshida (2019): *Less is more: multiparty session types revisited*. *Proc. ACM Program. Lang.* 3(POPL), pp. 30:1–30:29, doi:10.1145/3290343. Available at https://doi.org/10.1145/3290343.

[23] Philippe Schnoebelen (2002): *Verifying lossy channel systems has nonprimitive recursive complexity*. *Inf. Process. Lett.* 83(5), pp. 251–261, doi:10.1016/S0020-0190(01)00337-4. Available at https://doi.org/10.1016/S0020-0190(01)00337-4.

[24] Philippe Schnoebelen (2021): *On Flat Lossy Channel Machines*. In Christel Baier & Jean Goubault-Larrecq, editors: *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, *LIPIcs* 183, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 37:1–37:22, doi:10.4230/LIPIcs.CSL.2021.37. Available at https://doi.org/10.4230/LIPIcs.CSL.2021.37.

[25] Koushik Sen & Mahesh Viswanathan (2006): *Model Checking Multithreaded Programs with Asynchronous Atomic Methods*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 300–314.

[26] Salvatore La Torre, P. Madhusudan & Gennaro Parlato (2008): *Context-Bounded Analysis of Concurrent Queue Systems*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, *Lecture Notes in Computer Science* 4963, Springer, pp. 299–314, doi:10.1007/978-3-540-78800-3_21. Available at https://doi.org/10.1007/978-3-540-78800-3_21.

A Semantic Model for Interacting Cyber-Physical Systems

Benjamin Lion[1], Farhad Arbab[1,2], and Carolyn Talcott[3]

[1] Leiden University, Leiden, The Netherlands,
b.lion@liacs.leidenuniv.nl
[2] CWI, Amsterdam, The Netherlands
farhad@cwi.nl
[3] SRI International, CA, USA
carolyn.talcott@gmail.com

Abstract. We propose a component-based semantic model for Cyber-Physical Systems (CPSs) wherein the notion of a component abstracts the internal details of both cyber and physical processes, to expose a uniform semantic model of their externally observable behaviors expressed as sets of sequences of observations. We introduce algebraic operations on such sequences to model different kinds of component composition. These composition operators yield the externally observable behavior of their resulting composite components through specifications of interactions of the behaviors of their constituent components, as they, e.g., synchronize with or mutually exclude each other's alternative behaviors. Our framework is expressive enough to allow articulation of properties that coordinate desired interactions among composed components within the framework, also as component behavior. We demonstrate the usefulness of our formalism through examples of coordination properties in a CPS consisting of two robots interacting through shared physical resources.

1 Introduction

Compositional approaches in software engineering reduce the complexity of specification, analysis, verification, and construction of software by decomposing it down into (a) smaller parts, and (b) their interactions. Applied recursively, thus, compositional methods reduce software complexity by breaking the software and its parts down into ultimately simple modules, each with a description, properties, and interactions of manageable size. The natural tendency to regard each physical entity as a separate module in a Cyber-Physical System (CPS) makes compositional methods particularly appealing for specification, analysis, verification, and construction of CPSs. However, the distinction between discrete versus continuous transformations in modules representing cyber versus physical processes complicates the semantics of their specification and their treatment by requiring: (1) distinct formalisms to model discrete and continuous phenomena; (2) distinct formalisms to express composition and interactions of cyber-cyber, cyber-physical, and physical-physical pairs of modules; and (3) when to use which

formalism to express composition and interactions of hybrid cyber-physical modules.

Contribution

- we propose a semantic model of interacting cyber and physical processes based on sequences of observations,
- we define an algebraic framework to express interactions between time sensitive components,
- we give a general mechanism, using a co-inductive construction, to define algebraic operations on components as a lifting of some constraints on observations,
- we introduce two classes of properties on components, trace properties and behavior properties, and demonstrate their application in an example.

Our approach differs from more concrete approaches (e.g., operational models, executable specifications, etc.) in the sense that our operations on components model operations of composition at the semantic level.

We first intuitively introduce some key concepts and an example in Section 2. We provide in Section 3 formal definitions for components, their composition, and their properties. We describe a detailed example in Section 4. We present some related and our future work in Section 5, and conclude the paper in Section 6.

2 Coordination of energy-constrained robots on a field

In this work, we consider a cyber-physical system as a set of interacting processes. Whether a process consists of a physical phenomenon (sun rising, electrochemical reaction, etc.) or a cyber phenomenon (computation of a function, message exchanges, etc.), it exhibits an externally observable behavior resulting from some internal non-visible actions. Instead of a unified way to describe internals of cyber and physical processes, we propose a uniform description of what we can externally observe of their behavior and interactions.

In this section, we introduce some concepts that we will formalize later. An *event* may describe something like *the sun-rise* or *the temperature reading of* $5°C$. An event occurs at a point in time, yielding an event occurrence (e.g., the sun-rise event occurred at 6:28 am today), and the same event can occur repeatedly at different times (the sun-rise event occurs every day). Typically, multiple events may occur at "the same time" as measured within a measurement tolerance (e.g., the bird vacated the space at the same time as the bullet arrived there; the red car arrived at the middle of the intersection at the same time as the blue car did). We call a set of events that occur together at the same time an *observable*. A pair (O, t) of a set of observable events O together with its timestamp t represents an *observation*. An observation (O, t) in fact consists of a set of event occurrences: occurrences of events in O at the same time t. We call an infinite sequence of observations a *Timed-Event Stream* (TES). A *behavior* is a set of TESs. A *component* is a behavior with an interface.

Consider two robot components, each interacting with its own local battery component, sharing a field resource. The fact that the robots share the field through which they roam, forces them to somehow coordinate their (move) actions. Coordination is a set of constraints imposed on the otherwise possible observable behavior of components. In the case of our robots, if nothing else, at least physics prevents the two robots from occupying the same field space at the same time. More sophisticated coordination may be imposed (by the robots themselves or by some other external entity) to restrict the behavior of the robots and circumvent some undesirable outcomes, including hard constraints imposed by the physics of the field. The behaviors of components consist of timed-event streams, where events may include some measures of physical quantities. We give in the sequel a detailed description of three components, a robot (R), a battery (B), and a field (F), and of their interactions. We use SI system units to quantify physical values, with time in seconds (s), charging status in percentage (%), distance in meters (m), speed in meters per second ($\mathrm{m\,s^{-1}}$).

Table 1. Each column displays a segment of a timed-event stream for a robot, a battery, and a field component, where observables are singleton events. For $t \in \mathbb{R}_+$, we use $R(t), B(t)$, and $F(t)$ to respectively denote the observable at time t for the TES in the Robot, the Battery, and the Field column. An explicit empty set is not mandatory if no event is observed.

	Robot (R)	Battery (B)	Field (F)	Robot-Battery-Field
1s	$\{(read(loc, R); (0; 0))\}$		$\{(loc(i); (0; 0))\}$	$R(1) \cup F(1)$
2s	$\{(move(R); (N, 1\,\mathrm{m\,s^{-1}}))\}$	$\{(discharge(B); 1\%)\}$	$\{(move(I); (N, 0.5))\}$	$R(2) \cup B(2) \cup F(2)$
3s	$\{(read(loc, R); (0; 1))\}$		$\{(loc(I); (0; 1))\}$	$R(3) \cup F(3)$
4s	$\{(read(bat, R); 92\%)\}$	$\{(read(B); 92\%)\}$		$R(4) \cup B(4)$
...	...	...	...	...

A *robot* component, with identifier R, has two kinds of events: read events $(read(bat, R); b)$ that measures the level b of its battery or $(read(loc, R); l)$ that obtains its position l, and a move event $(move(R); (d, v))$ when the robot moves in the direction d with speed v. The TES in the Robot column in Table 1 shows a scenario where robot R reads its location and gets the value $(0; 0)$ at time 1s, then moves north at one meter per second at time 2s, reads its location and gets $(0; 1)$ at time 3s, and reads its battery value and gets 92% at time 4s, ...

A *battery* component, with identifier B, has three kinds of events: a charge event $(charge(B); \eta_2)$, a discharge event $(discharge(B); \eta_1)$, and a read event $(read(B); s)$, where η_1 and η_2 are respectively the discharge and charge factor of the battery, and s is the current charge status. The TES in the Battery column in Table 1 shows a scenario where the battery discharged at a rate of 1% per second at time 2s, and reported its charge-level of 92% at time 4s, ...

A *field* component, with identifier F, has two kinds of events: a position event $(loc(I); p)$ that obtains the position p of an object I, and a move event $(move(I); (d, \eta))$ of the object I in the direction d with a friction factor η. The TES in the Field column in Table 1 shows a scenario where the field has the object I at location $(0; 0)$ at time 1s, then the object I moves in the north direction with a friction coefficient of 0.5 at time 2s, subsequently to which the object I is at location $(0; 1)$ at time 3s, ...

When components interact with each other, in a shared environment, behaviors in their composition must also compose with a behavior of the environment. For instance, a battery component may constrain how many amperes it delivers, and therefore restrict the speed of the robot that interacts with it. We specify interaction explicitly as an exogenous binary operation that constrains the composable behaviors of its operand components.

The *robot-battery* interaction imposes that a move event in the behavior of a robot coincides with a discharge event in the behavior of the robot's battery, such that the discharge factor of the battery is proportional to the speed of the robot. The physicality of the battery prevents the robot from moving if the energy level of the battery is not sufficient (i.e., such an anomalous TES would not exist in the battery's behavior, and therefore cannot compose with a robot's behavior). Moreover, a read event in the behavior of a robot component should also coincide with a read event in the behavior of its corresponding battery component, such that the two events contain the same charge value.

The *robot-field* interaction imposes that a move event in the behavior of a robot coincides with a move event of an object on the field, such that the friction coefficient on the field is proportional to the speed of the robot. A read event in the behavior of a robot coincides with a position event of the corresponding robot object on the field, such that the two events contain the same position value. Additional interaction constraints may be imposed by the physics of the field. For instance, the constraint "no two robots can be observed at the same location" would rule out every behavior where the two robots are observed at the same location.

A TES for the composite Robot-Battery-Field system collects, in sequence, all observations from a TES in a Robot, a Battery, and a Field component behavior, such that at any moment the interaction constraints are satisfied. The column Robot-Battery-Field in Table 1 displays the first elements of such a TES.

3 Components, composition, and properties

3.1 Notations

An *event* is a simplex (the most primitive form of an) observable element. An event may or may not have internal structure. For instance, the successive ticks of a clock are occurrences of a tick event that has no internal structure; successive readings of a thermometer, on the other hand, constitute occurrences of a temperature-reading event, each of which has the internal structure of a name-value pair . Similarly, we can consider successive transmissions by a mobile sensor as occurrences of a structured event, each instance of which includes geolocation coordinates, barometric pressure, temperature, humidity, etc. Regardless of whether or not events have internal structures, in the sequel, we regard events as uninterpreted simplex observable elements.

Notation 1 (Events) *We use* $\mathbb{E}$ *to denote the universal set of events.*

An *observable* is a set of event occurrences that happen together and an *observation* is a pair (O, t) of an observable O and a time-stamp $t \in \mathbb{R}_+$.[1] An observation (O, t) represents an act of atomically observing occurrences of events in O at time t. Atomically observing occurrences of events in O at time t means there exists a small $\epsilon \in \mathbb{R}_+$ such that during the time interval $[t - \epsilon, t + \epsilon]$:

1. every event $e \in O$ is observed exactly once[2], and
2. no event $e \notin O$ is observed.

We write $\langle s_0, s_1, ..., s_{n-1} \rangle$ to denote a *finite sequence of size* n of elements over an arbitrary set S, where $s_i \in S$ for $0 \leq i \leq n-1$. The set of all finite sequences of elements in S is denoted as S^*. A *stream*[3] over a domain S is a function $\sigma : \mathbb{N} \to S$. We use $\sigma(i)$ to represent the $i+1^{st}$ element of σ, and given a finite sequence $s = \langle s_0, ..., s_{n-1} \rangle$, we write $s \cdot \sigma$ to denote the stream $\tau \in \mathbb{N} \to S$ such that $\tau(i) = s_i$ for $0 \leq i \leq n-1$ and $\tau(i) = \sigma(i-n)$ for $n \leq i$. We use σ' to denote the derivative of σ, such that $\sigma'(i) = \sigma(i+1)$ for all $i \in \mathbb{N}$.

A *Timed-Event Stream (TES)* over a set of events E and a set of time-stamps $\mathbb{R}_+$ is a stream $\sigma \in \mathbb{N} \to (\mathcal{P}(E) \times \mathbb{R}_+)$ where, for $\sigma(i) = (O_i, t_i)$:

1. for every $i \in \mathbb{N}, t_i < t_{i+1}$, [i.e., time monotonically increases] and
2. for every $n \in \mathbb{N}$, there exists $i \in \mathbb{N}$ such that $t_i > n$ [i.e., time is non-Zeno progressive].

Notation 2 (Time stream) *We use $OS(\mathbb{R}_+)$ to refer to the set of all monotonically increasing and non-Zeno infinite sequences of elements in $\mathbb{R}_+$.*

Notation 3 (Timed-Event Stream) *We use $TES(E)$ to denote the set of all TESs whose observables are subsets of the event set E with elements in $\mathbb{R}_+$ as their time-stamps.*

Given a sequence $\sigma = \langle (O_0, t_0), (O_1, t_1), (O_2, t_2), ... \rangle \in TES(E)$, we use the projections $\mathsf{pr}_1(\sigma) \in \mathbb{N} \to \mathcal{P}(E)$ and $\mathsf{pr}_2(\sigma) \in \mathbb{N} \to \mathbb{R}_+$ to denote respectively the sequence of observables $\langle O_0, O_1, O_2, ... \rangle$ and the sequence of time stamps $\langle t_0, t_1, t_2, ... \rangle$.

3.2 Components

The design of complex systems becomes simpler if such systems can be decomposed into smaller sub-systems that interact with each other. In order to simplify the design of cyber-physical systems, we abstract from the internal details of both cyber and physical processes, to expose a uniform semantic model. As a first class entity, a component encapsulates a behavior (set of TESs) and an interface (set of events).

[1] Any totally ordered dense set would be suitable as the domain for time (e.g., positive rationals $\mathbb{Q}_+$). For simplicity, we use $\mathbb{R}_+$, the set of real numbers $r \geq 0$ for this purpose.

[2] A finer time granularity may reveal some ordering relation on the occurrence of events in the same set of observation.

[3] The set $\mathbb{N}$ denotes the set of natural numbers $n \geq 0$.

Definition 1 (Component). *A component is a tuple $C = (E, L)$ where $E \subseteq \mathbb{E}$ is a set of events, and $L \subseteq TES(E)$ is a set of TESs. We call E the* interface *and L the externally observable* behavior *of C.*

More particularly, Definition 1 makes no distinction between cyber and physical components. We use the following example to describe some cyber and physical aspects of components.

Example 1. Consider a component encapsulating a continuous function $f : (D_0 \times \mathbb{R}_+) \to D$, where D_0 is a set of initial values, and D is the codomain of values for f. Such a function can describe the evolution of a physical system over time, where $f(d_0, t) = d$ means that at time t the state of the system is described by the value $d \in D$ if initialized with d_0. We define the set of all events for this component as the range of function f given an initial parameter $d_0 \in D_0$. The component is then defined as the pair (D, L_f) such that:

$$L_f = \{\sigma \in TES(D) \mid \exists d_0 \in D_0.\ \forall i \in \mathbb{N}.\ \mathsf{pr}_1(\sigma)(i) = \{f(d_0, \mathsf{pr}_2(\sigma)(i))\}\}$$

Observe that the behavior of this component contains all possible discrete samplings of the function f at monotonically increasing and non-Zeno sequences of time stamp. Different instances of f would account for various cyber and physical aspects of components. ■

3.3 Composition

A complex system typically consists of multiple components that interact with each other. The example in Section 2 shows three components, a *robot*, a *battery*, and a *field*, where: a move observable of a robot must coincide with a move observable of the field and a discharge observable of its battery.

We express such constraints on behaviors using relations[4]. It is sometimes necessary to relate TESs of two components to express what must not, but otherwise may, happen. Sometimes whether or not a pair of observations in two TESs can compose, depends on the events involved in those observations. To capture this notion, we introduce a generalized notion of a *composability relation* as a parametrized relation that takes as argument a pair of carrier sets of events and relates pairs of TESs over those event sets.

Definition 2 (Composability relation on TESs). *A composability relation is a parametrized relation R such that for all $E_1, E_2 \subseteq \mathbb{E}$, we have $R(E_1, E_2) \subseteq TES(E_1) \times TES(E_2)$.*

Definition 3 (Symmetry). *A parametrized relation Q is* symmetric *if, for all (x_1, x_2) and for all (X_1, X_2): $(x_1, x_2) \in Q(X_1, X_2) \iff (x_2, x_1) \in Q(X_2, X_1)$.*

[4] Also non binary relations could be considered, i.e., constraints imposed on two components.

A composability relation on TESs serves as a necessary constraint for two TESs to compose. We give in Section 3.4 some examples of useful composability relations on TESs that, e.g., enforce synchronization or mutual exclusion of observables. We define *composition* of TESs as the act of forming a new TES out of two TESs.

Definition 4. *A composition function $\oplus$ on TES is a function $\oplus : TES(\mathbb{E})^2 \to TES(\mathbb{E})$.*

We define a binary product operation on components, parametrized by a composability relation and a composition function on TESs, that forms a new component. Intuitively, the newly formed component describes, by its behavior, the evolution of the joint system under the constraint that the interactions in the system satisfy the composability relation. Formally, the product operation returns another component, whose set of events is the union of sets of events of its operands, and its behavior is obtained by composing all pairs of TESs in the behavior of its operands deemed composable by the composability relation.

Definition 5 (Product). *Let $(R, \oplus)$ be a pair of a composition function and a composability relation on TESs, and $C_i = (E_i, L_i)$, $i \in \{1, 2\}$, two components. The product of C_1 and C_2, under R and $\oplus$, denoted as $C_1 \times_{(R,\oplus)} C_2$, is the component (E, L) where $E = E_1 \cup E_2$ and L is defined by*

$$L = \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ (\sigma_1, \sigma_2) \in R(E_1, E_2)\}$$

Definition 5 presents a generic composition operator, where composition is parametrized over a composability relation and a composition function.

Lemma 1. *Let $\oplus_1$ and $\oplus_2$ be two composition functions on TESs, and let R_1 and R_2 be two composability relations on TESs. Then:*

- *if R_1 is symmetric, then $\times_{(R_1,\oplus_1)}$ is* commutative *if and only if $\oplus_1$ is commutative;*
- *if, for all $E_i \subseteq \mathbb{E}$ and $\sigma_i \in TES(E_i)$ with $i \in \{1, 2, 3\}$, we have*

$$(\sigma_1, \sigma_2 \oplus_2 \sigma_3) \in R_1(E_1, E_2 \cup E_3) \wedge (\sigma_2, \sigma_3) \in R_2(E_2, E_3) \iff$$
$$(\sigma_1, \sigma_2) \in R_1(E_1, E_2) \wedge (\sigma_1 \oplus_1 \sigma_2, \sigma_3) \in R_2(E_1 \cup E_2, E_3)$$

  then $\times_{(R_1,\oplus_1)}$ and $\times_{(R_2,\oplus_2)}$ are associative *if and only if $\sigma_1 \oplus_1 (\sigma_2 \oplus_2 \sigma_2) = (\sigma_1 \oplus_1 \sigma_2) \oplus_2 \sigma_3$*
- *if for all $E \subseteq \mathbb{E}$ and $\sigma, \tau \in TES(E)$, we have $(\sigma, \tau) \in R_1(E, E) \implies \sigma = \tau$, then $\times_{(R_1,\oplus_1)}$ is* idempotent *if and only if $\oplus_1$ is idempotent.*

The generality of our formalism allows exploration of other kinds of operations on components, such as division. Intuitively, the division of a component C_1 by a component C_2 yields a component C_3 whose behavior contains all TESs that can compose with TESs in the behavior of C_2 to yield the TESs in the behavior of C_1.

Definition 6 (Division). *Let R be a composability relation on TESs, and $\oplus$ a composition function on TESs. The division of two components $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$ under R and $\oplus$, denoted as $C_1/_{(R,\oplus)}C_2$, is the component $C = (E_1, L)$ such that:*

$$L = \{\sigma \in TES(E_1) \mid \exists \sigma_2 \in L_2.\ (\sigma, \sigma_2) \in R(E_1, E_2) \wedge \sigma \oplus \sigma_2 \in L_1\}$$

If the dividend is $C_1 = C_1' \times_{(R,\oplus)} C_2'$, and the divisor is an operand of the product, e.g., $C_2 = C_2'$, then the behavior of the result of the division, C, contains all TESs in the behavior of the other operand (i.e., C_1') composable with a TES in the behavior of C_2.

Lemma 2. *Let $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$ be two components. Let $(C_1 \times_{(R,\oplus)} C_2)/_{(R,\oplus)}C_2 = (E_3, L_3)$, with $(R, \oplus)$ a pair of a composability relation and a composition function on TESs. Then,*

$$\{\sigma_1 \in L_1 \mid \exists \sigma_2 \in L_2.\ (\sigma_1, \sigma_2) \in R(E_1 \cup E_2, E_2) \cap R(E_1, E_2)\} \subseteq L_3$$

Corollary 1. *In the case where $R = \top$ (see Definition 10) then $L_1 \subseteq L_3$.*

3.4 A co-inductive construction for composability relations

In this section, we provide a co-inductive construction for composability relations on TESs. We show how some constraints on observations can be *lifted* to constraints on TESs, and give weaker conditions for Lemma 1 to hold.

Intuitively, the lifting of a composition function from observables to TESs compares element wise the observations of two TESs, and distinguishes three cases. Two symmetric cases occur when the first observation of the two TESs do not occur at the same time: the earliest observation is added in sequence to the resulting TES. In the case where the first observations of the two TESs have the same time stamp, a newly formed observation is added to the resulting TES, whose time stamp is the same and whose observable is obtained by composition of the two observables.

Definition 7 (Lifting - composition function). *Let $+ : \mathcal{P}(\mathbb{E})^2 \to \mathcal{P}(\mathbb{E})$ be a composition function on observables. The* lifting *of $+$ to TESs is $[+] : TES(\mathbb{E})^2 \to TES(\mathbb{E})$ s.t., for $\sigma_i \in TES(\mathbb{E})$ where $\sigma_i(0) = (O_i, t_i)$ with $i \in \{1, 2\}$:*

$$\sigma_1[+]\sigma_2 = \begin{cases} \langle \sigma_1(0) \rangle \cdot (\sigma_1'[+]\sigma_2) & \text{if } t_1 < t_2 \\ \langle \sigma_2(0) \rangle \cdot (\sigma_1[+]\sigma_2') & \text{if } t_2 < t_1 \\ \langle (O_1 \oplus O_2, t_1) \rangle \cdot (\sigma_1'[+]\sigma_2') & \text{otherwise} \end{cases}$$

Definition 7 requires two observations to have the exact same time stamp to compose. Alternative definitions are also possible where the timing constraint is relaxed to time intervals instead of exact times.

Similarly, we introduce composability relations on observations, and give a mechanism to lift such composability relations to relate TESs.

Definition 8 (Composability relation on observations). *A composability relation on observations is a parametrized relation κ s.t. for all pairs $(E_1, E_2) \in \mathcal{P}(\mathbb{E})^2$, we have $\kappa(E_1, E_2) \subseteq (\mathcal{P}(E_1) \times \mathbb{R}_+) \times (\mathcal{P}(E_2) \times \mathbb{R}_+)$*

Definition 9 (Lifting- composability relation). *Let κ be a composability relation on observations, and let $\Phi_\kappa : \mathcal{P}(\mathbb{E})^2 \to (\mathcal{P}(TES(\mathbb{E})^2) \to \mathcal{P}(TES(\mathbb{E})^2))$ be such that, for any $\mathcal{R} \subseteq TES(\mathbb{E})^2$:*

$$\begin{aligned}\Phi_\kappa(E_1, E_2)(\mathcal{R}) = \{(\tau_1, \tau_2) \mid\ & (\tau_1(0), \tau_2(0)) \in \kappa(E_1, E_2) \wedge \\ & (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ & (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R} \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R} \vee \\ & \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R})\}\end{aligned}$$

The lifting *of κ on TESs, written $[\kappa]$, is the parametrized relation obtained by taking the fixed point of the function $\Phi_\kappa(E_1, E_2)$ for arbitrary pair $E_1, E_2 \subseteq \mathbb{E}$, i.e., the relation $[\kappa](E_1, E_2) = \bigcup_{\mathcal{R} \subseteq TES(\mathbb{E}) \times TES(\mathbb{E})} \{\mathcal{R} \mid \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R})\}$.*

Lemma 3 (Correctness of lifting). *For any $E_1, E_2 \subseteq \mathbb{E}$, the function Φ_κ is monotone, and therefore has a greatest fixed point.*

Lemma 4. *If κ is a composability relation on observations, then the lifting $[\kappa]$ is a composability relation on TESs. Moreover, if κ is symmetric (as in Definition 3), then $[\kappa]$ is symmetric.*

We give three examples of composability relation on TESs, where Definition 11 and Definition 12 are two examples that construct co-inductively the composability relation on TESs from a composability relation on observations. For the following examples, let $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$ be two components, and $\oplus$ be a composition function on TESs.

Definition 10 (Free composition). *We use $\top$ for the most permissive composability relation on TESs such that, for any $\sigma, \tau \in TES(\mathbb{E})$, we have $(\sigma, \tau) \in \top$.*

The behavior of component $C_1 \times_{(\top, \oplus)} C_2$ contains every TES obtained from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs. This product does not impose any constraint on event occurrences of its operands.

Definition 11 (Synchronous composition). *Let $\sqcap \subseteq \mathcal{P}(\mathbb{E})^2$ be a composability relation on observables. We define two observations to be synchronous under the composability relation $\sqcap$ according to the following two conditions:*

1. *every observable that can compose (under $\sqcap$) with another observable must occur simultaneously with one of its composable observables; and*
2. *only an observable that does not compose (under $\sqcap$) with any other observable can occur independently, i.e., at a different time.*

A synchronous *composability relation on observations $\kappa_{sync,\sqcap}(E_1, E_2)$ satisfies the two conditions above. For any two observations $(O_i, t_i) \in \mathcal{P}(E_i) \times \mathbb{R}_+$ with $i \in \{1, 2\}$, $((O_1, t_1), (O_2, t_2)) \in \kappa_{sync,\sqcap}(E_1, E_2)$ if and only if:*

$$\begin{aligned}&(t_1 < t_2 \wedge \neg(\exists O_2' \subseteq E_2.\ (O_1, O_2') \in \sqcap)) \ \vee (t_2 < t_1 \wedge \neg(\exists O_1' \subseteq E_1.\ (O_1', O_2) \in \sqcap)) \vee \\ &\ t_2 = t_1 \ \wedge ((O_1, O_2) = (O_1' \cup O_1'', O_2' \cup O_2'') \wedge (O_1', O_2') \in \sqcap \wedge \\ &\qquad (\forall O \subseteq E_2.\ (O_1'', O) \notin \sqcap) \wedge \forall O \subseteq E_1.\ (O, O_2'') \notin \sqcap)) \vee (O_1, O_2) = (\emptyset, \emptyset))\end{aligned}$$

Example 2. Let $E_1 = \{a, b\}$ and $E_2 = \{c, d\}$ with $\sqcap = \{(\{a\}, \{c\})\}$. Thus, $((\{a\}, t_1), (\{d\}, t_2)) \in \kappa_{sync,\sqcap}$ if and only if $t_2 < t_1$. Alternatively, we have $((\{a\}, t_1), (\{c\}, t_2)) \in \kappa_{sync,\sqcap}$ if and only if $t_1 = t_2$.

The lifting $[\kappa_{sync,\sqcap}]$, written $\bowtie_\sqcap$, defines a synchronous composability relation on TESs. The behavior of component $C_1 \times_{(\bowtie_\sqcap,\oplus)} C_2$ contains TESs obtained from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs that are related by the synchronous composability relation $\bowtie_\sqcap$ which, depending on $\sqcap$, may exclude some event occurrences unless they synchronize. [5]

Definition 12 (Mutual exclusion). *Let $\sqcap \subseteq \mathcal{P}(\mathbb{E})^2$ be a composability relation on observables. We define two observations to be mutually exclusive under the composability relation $\sqcap$ if no pair of observables in $\sqcap$ can be observed at the same time. The mutually exclusive composability relation $\kappa_{excl,\sqcap}$ on observations allows the composition of two observations (O_1, t_1) and (O_2, t_2), i.e., $((O_1, t_1), (O_2, t_2)) \in \kappa_{excl,\sqcap}(E_1, E_2)$, if and only if*

$$(t_1 < t_2) \vee (t_2 < t_1) \vee (\neg(O_1 \sqcap O_2) \wedge t_1 = t_2)$$

Example 3. Let $E_1 = \{a, b\}$ and $E_2 = \{c, d\}$ with $\sqcap = \{(\{a\}, \{c\})\}$. Thus, $((\{a\}, t_1), (\{c\}, t_2)) \notin \kappa_{excl,\sqcap}$ for any $t_2 = t_1$, and $\{a\}$ and $\{c\}$ are two mutually exclusive observables.

Similarly as in Example 11, the lifting $[\kappa_{excl,\sqcap}]$ of $\kappa_{excl,\sqcap}$, written $\nparallel_\sqcap$, defines a mutual exclusion composability relation on TESs. The behavior of component $C_1 \times_{(\nparallel_\sqcap,\oplus)} C_2$ contains TESs resulting from the composition under $\oplus$ of every pair $\sigma_1 \in L_1$ and $\sigma_2 \in L_2$ of TESs that are related by the mutual exclusion composability relation $\nparallel_\sqcap$ which, depending on $\sqcap$, may exclude some simultaneous event occurrences.

Besides the product instances detailed in Definitions 10, 11, 12, the definition of composability relation or composition function as the lift of some relations on observations or function on observables allows weaker sufficient conditions for Lemma 1 to hold.

Lemma 5. *Let $+_1$ and $+_2$ be two composition functions on observables and let κ_1 and κ_2 be two composability relations on observations. Then,*

- $\times_{([\kappa_1],[+_1])}$ *is* commutative *if κ_1 is symmetric and $+_1$ is commutative;*
- $\times_{([\kappa_1],[+_1])}$ *and* $\times_{([\kappa_2],[+_2])}$ *are* associative *if, for all $E_i \subseteq \mathbb{E}$ and for any triple of observations $o_i = (O_i, t_i) \in \mathcal{P}(E_i) \times \mathbb{R}_+$ with $i \in \{1, 2, 3\}$, we have $(O_1 +_1 O_2) +_2 O_3 = O_1 +_1 (O_2 +_2 O_3)$ and*

$$\begin{aligned}&((o_1, o_2) \in \kappa(E_1, E_2) \wedge (\iota_1(o_1, o_2), o_3) \in \kappa(E_1 \cup E_2, E_3)) \iff \\ &((o_2, o_3) \in \kappa(E_2, E_3) \wedge (o_1, \iota_2(o_2, o_3)) \in \kappa(E_1, E_2 \cup E_3))\end{aligned}$$

[5] If we let $\oplus$ be the element wise set union, define an event as a set of port assignments, and in the pair $(\bowtie_\sqcap, \oplus)$ let $\sqcap$ be true if and only if all common ports get the same value assigned, then this composition operator produces results similar to the composition operation in Reo [4].

$$\text{with } \iota_k((O,t),(P,l)) = \begin{cases} (O,t) & \text{if } t < l \\ (P,l) & \text{if } l < t \\ (O +_k P, t) & \end{cases} \text{, where } k \in \{1,2\}.$$

– $\times_{([\kappa_1],[+_1])}$ *is* idempotent *if* $+_1$ *is idempotent and, for all* $E \subseteq \mathbb{E}$ *we have* $((O_1,t_1),(O_2,t_2)) \in \kappa_1(E,E) \implies (O_1,t_1) = (O_2,t_2)$.

3.5 Properties

We distinguish two kinds of properties: properties on TESs that we call *trace properties*, and properties on sets of TESs that we call *behavior properties*, which correspond to hyper-properties in [6]. The generality of our model permits to interchangeably construct a component from a property and extract a property from a component. As illustrated in Example 5, when composed with a set of interacting components, a component property constrains the components to only expose desired behavior (i.e., behavior in the property). In Section 4, we provide more intuition for the practical relevance of these properties.

Definition 13. *A trace property* P *over a set of events* E *is a subset* $P \subseteq TES(E)$. *A component* $C = (E, L)$ *satisfies a property* P, *if* $L \subseteq P$, *which we denote as* $C \models P$.

Example 4. A trace property is similar to a component, since it describes a set of TESs, except that it is *a priori* not restricted to any interface. A trace property P can then be turned into a component, by constructing the smallest interface E_P such that, for all $\sigma \in P$, and $i \in \mathbb{N}$, $\mathsf{pr}_1(\sigma)(i) \subseteq E_P$. The component $C_P = (E_P, P)$ is then the componentized-version of property P. ■

Lemma 6. *Let* E *be a set of events, and let* $\sqcap$ *be the smallest relation such that for all non empty* $O \subseteq E$, $(O,O) \in \sqcap$. *Given a property* P *over* E, *its componentized-version* C_P *(see Example 4), the product* $\bowtie_\sqcap$ *as in Definition 11, and a component* $C = (E, L)$, *then* $C \models P$ *if and only if* $C \times_{(\bowtie_\sqcap,[\cup])} C_P = C$.

Example 5. We use the term *coordination property* to refer to a property used in order to coordinate behaviors. Given a set of n components $C_i = (E_i, L_i)$, $i \in \{1, ..., n\}$, a coordination property *Coord* ranges over the set of events $E = E_1 \cup ... \cup E_n$, i.e., $Coord \subseteq TES(E)$.

Consider the synchronous interaction of the n components, with $\sqcap$ a symmetric composability relation on observables, $\oplus$ an associative and commutative composition function on TESs, and let $C = ((C_1 \times_{(\bowtie_\sqcap,\oplus)} C_2) ... \times_{(\bowtie_\sqcap,\oplus)} C_n)$ be their synchronous product (see Corollary 2 in Appendix 7 for associativity of $\times_{(\bowtie_\sqcap,\oplus)}$). Typically, a coordination property will not necessarily be satisfied by the composite component C, but some of the behavior of C is contained in the coordination property. The coordination problem is to find (e.g., synthesize) an orchestrator component $Orch = (E_O, L_O)$ such that $C \times_{(\bowtie_\sqcap,\oplus)} Orch \models Coord$. The orchestrator restricts the component C to exhibit only the subset of its behavior that satisfies the coordination property. In other words, in their composition, *Orch* coordinates C to satisfy *Coord*. The coordination problem can

be made even more granular by changing the composability relations or the composition functions used in the construction of C.

As shown in Example 4, since *Coord* ranges over the same set E that is the interface of component $((C_1 \times_{(\bowtie_\sqcap, \oplus)} C_2) ... \times_{(\bowtie_\sqcap, \oplus)} C_n)$, a coordination property can be turned into an orchestrator by building its corresponding component. ■

Trace properties are not sufficient to fully capture the scope of interesting properties of components and cyber-physical systems. Some of their limitations are highlighted in Section 4. To address this issue, we introduce *behavior properties*, which are strictly more expressive than trace properties, and give two illustrative examples.

Definition 14. *A behavior property ϕ over a set of events E is a hyper-property $\phi \subseteq \mathcal{P}(TES(E))$. A component $C = (E, L)$ satisfies a hyper-property ϕ if $L \in \phi$, which we denote as $C \Vdash \phi$.*

Example 6. A component $C = (E, L)$ can be oblivious to time. Any sequence of time-stamps for an acceptable sequence of observables is acceptable in the behavior of such a component. This "obliviousness to time" property is not a trace property, but a hyper-property, defined as:

$$\phi_{shift}(E) := \{Q \subseteq TES(E) \mid \forall \sigma \in Q. \forall t \in OS(\mathbb{R}_+). \exists \tau \in Q. \mathsf{pr}_1(\sigma) = \mathsf{pr}_1(\tau) \wedge \mathsf{pr}_2(\tau) = t\}$$

Intuitively, if $C \Vdash \phi_{shift}(E)$, then C is independent of time. ■

Example 7. We use $\phi_{insert}(X, E)$ to denote the hyper-property that allows for arbitrary insertion of observations in $X \subseteq \mathcal{P}(E)$ into every TES at any point in time, i.e., the set defined as:

$$\begin{aligned} \{Q \subseteq TES(E) \mid \forall \sigma \in Q. \forall i \in \mathbb{N}. \exists \tau \in Q. \exists x \in X. (&\forall j < i. \quad \sigma(j) \;\; = \tau(j)) \wedge \\ &(\exists t \in \mathbb{R}_+. \quad \tau(i) \;\; = (x, t)) \wedge \\ &(\forall j \geq i. \quad \tau(j+1) = \sigma(j)) \;\} \end{aligned}$$

Intuitively, elements of $\phi_{insert}(X, E)$ are closed under insertion of an observation $O \subseteq X$ at an arbitrary time. ■

4 Application

This section is inspired from the work on soft-agents [19, 10], and elaborates on the more intuitive version presented in Section 2. We show in subsections 4.1 and 4.2 some expressions that represent interactive cyber-physical systems, and in subsection 4.3 we formulate some trace and behavior properties on those systems.

4.1 Description of components

We give, in order, a description for a robot, a battery, and a field component.

Robot. A robot component R is a tuple (E_R, L_R) with:

$$E_R = \{(read(loc, R); l), (read(bat, R); b), (move(R); (d, v)), (charge(R); s) \mid \\ l \in [0, 20]^2,\ 0 \leq b \leq 100\%,\ d \in \{\mathsf{N}, \mathsf{E}, \mathsf{W}, \mathsf{S}\},\ v \in \mathbb{R}_+,\ s \in \{\mathsf{ON}, \mathsf{OFF}\}\}$$
$$L_R = \{\sigma \in TES(E_R) \mid \forall i \in \mathbb{N}. \exists e \in E_R.\ \mathsf{pr}_1(\sigma)(i) = \{e\}\}$$

where the read of the position, the read of the battery, the move, and the charge events contain respectively the position of the robot as a pair of coordinates on a $[0, 20]^2$ grid; the battery level as a percentage; the move direction as pair of a cardinal direction and a positive number for speed; and the charge status as ON or OFF. Note that the set of TESs L_R enforces no timing constraints.

Battery. A battery component B is a tuple (E_B, L_B) with:

$$E_B = \{(read(B); l), (discharge(B); \eta_1), (charge(B); \eta_2) \mid 0 \leq l \leq 100\%,\ \eta_2 \in [0, 1], \\ \eta_1 : \mathbb{R}_+ \to [0, 1]\}$$
$$L_B = \{\sigma \in TES(E_B) \mid \forall i \in \mathbb{N}. \exists e \in E_B.\ \mathsf{pr}_1(\sigma)(i) = \{e\} \wedge P_B(\sigma)\}$$

where the read, the charge, and the discharge events respectively contain the current charge percentage, the discharge factor, and the charge factor. P_B is a safety property that enforces every behavior of the battery to satisfy a physical constraint. In our case, the property P_B requires every read event occurrence to return a value that depends on the time stamp of the observation and the previous sequence of observables. The property P_B assumes that initially at $t = 0$ the battery is at 100% charge, that the battery level decreases after each discharge event, increases after each charge event proportionally, respectively by the discharge and the charge factors, and prevents a discharge event from occurring if the level of the battery is 0%.

Field. A field component $F(l_0)$ contains a single object that we identify as I initially at location l_0, has a fixed size of $[0, 20]^2$, and contains a charging station at location $(5; 5)$. A field component is a tuple (E_F, L_F) with:

$$E_F = \{(loc(I); p), (move(I); (d, \eta)) \mid p \in [0, 20]^2,\ d \in \{\mathsf{N}, \mathsf{S}, \mathsf{E}, \mathsf{W}\},\ \eta : \mathbb{R}_+ \to [0, 1]\}$$
$$L_F = \{\sigma \in TES(E_F) \mid \forall i \in \mathbb{N}. \exists e \in E_F.\ \mathsf{pr}_1(\sigma)(i) = \{e\} \wedge P_F(\sigma)\}$$

where the loc and the move events respectively contain the position of object I and the pair of friction η and direction d of the move of object I. The friction factor η is proportional to the speed at which an object moves, and reflects the physical aspect of a field component. P_F is a safety property that enforces every TES to respect the physics of the field component. P_F models the case where the object I is initially at position $(0, 0)$ and every move event changes continuously the location of the object on the field according to the direction d, the speed v of the object, and the friction η. A move event has no effect if it occurs while the position of I is on the boundary of the field: it simulates the case of a fence, where moving against the fence would have the same observable as not moving. Alternatively, one can imagine a different structure for the field component, and change P_F accordingly.

Robots R_1 and R_2 are two instances of the robot component, where all occurrences of R have been renamed respectively to R_1 and R_2 (e.g., $(\mathit{read}(\mathit{loc}, R), l)$ becomes $(\mathit{read}(\mathit{loc}, R_1), l)$ for the robot instance R_1, etc.). Similarly, we consider B_1 and B_2 to be two instances of the battery component B, and $F_1((0;0))$ and $F_2((5;0))$ to be two instances of the field component F parametrized by the initial location for the object I, where the objects in fields F_1 and F_2 are renamed to I_1 and I_2, and respectively initialized at position $(0;0)$ and $(5;0)$.

4.2 Interaction

We detail three points of interactions on observables among a robot and its battery, a robot and a field on which it moves, and two instances of a field component.

Robot-battery. Interactions between a robot component and its battery are such that, for instance, every occurrence of a move event at the robot component must be simultaneous with a discharge event of the battery, with the discharge factor proportional to the speed of the robot. Given a robot component R and a battery component B, we define the symmetric composability relation $\sqcap_{RB}$ on the set $E_R \cup E_B$ to be the smallest relation such that:

$$\begin{array}{rll} \{(\mathit{read}(\mathit{bat}, R); b)\} \sqcap_{RB} \{(\mathit{read}(B); b)\} & & \text{for all } 0 \leq b \leq 100\% \\ \{(\mathit{move}(R); (d, v))\} \sqcap_{RB} \{(\mathit{discharge}(B); \eta_1(v))\} & & \text{for all } d \in \{\mathsf{N}, \mathsf{S}, \mathsf{W}, \mathsf{E}\}, v \in \mathbb{R}_+ \\ \{(\mathit{charge}(R); \mathsf{ON})\} \sqcap_{RB} \{(\mathit{charge}(B); \eta_2)\} & & \end{array}$$

Robot-field. Interactions between a robot component and a field component are such that, for instance, every move event of the robot component must be simultaneous with a move event of the object I on the field, with a friction factor proportional to the speed of the robot. Given a robot component R and a field component F, we define the symmetric composability relation $\sqcap_{RF}$ on the set $E_R \cup E_F$ to be the smallest relation such that:

$$\begin{array}{rll} \{(\mathit{read}(\mathit{loc}, R); l)\} \sqcap_{FR} \{(\mathit{loc}(I); l)\} & & \text{for all } l \in [0, 20]^2 \\ \{(\mathit{move}(R); (d, v))\} \sqcap_{FR} \{(\mathit{move}(I); (d, \eta(v)))\} & & \text{for all } d \in \{\mathsf{N}, \mathsf{W}, \mathsf{E}, \mathsf{S}\}, v \in \mathbb{R}_+ \\ \{(\mathit{charge}(R); \mathsf{ON})\} \sqcap_{FR} \{(\mathit{loc}(I); (5, 5))\} & & \end{array}$$

Observe that a robot can charge only if it is located at the charging station.

Field-field. We add also interaction constraints between two fields, such that no observation can gather two read events containing the same position value. Thus, given two fields F_1 and F_2, let $\sqcap_{F_{12}}$ be the smallest symmetric mutual exclusion composability relation on the set $E_{F_1} \cup E_{F_2}$ such that: $\{(\mathit{loc}(I_1); l)\} \sqcap_{F_{12}} \{(\mathit{loc}(I_2); l)\}$ for all $l \in [0, 20]^2$. Observe that we interpret $\sqcap_{F_{12}}$ as a mutual exclusion relation.

Product. We use set union as composition function on observables: given two observables O_1 and O_2, we define $O_1 \oplus O_2$ to be the observable $O_1 \cup O_2$. We use the synchronous and mutual exclusion composability relations on TESs introduced in Definition 11 and Definition 12. We represent the cyber-physical system

consisting of two robots R_1 and R_2 with two private batteries B_1 and B_2, and individual fields F_1 and F_2, as the expression:

$$F_1F_2 \times_{(\bowtie \sqcap_{FR_{12}}, [\cup])} (R_1B_1 \times_{(\top, [\cup])} R_2B_2) \tag{1}$$

where $F_1F_2 := (F_1 \times_{(\nparallel \sqcap_{F_{12}}, [\cup])} F_2)$, $R_iB_i := (R_i \times_{(\bowtie \sqcap_{R_iB_i}, [\cup])} B_i)$, and $\sqcap_{FR_{12}} := (\sqcap_{F_1R_1} \cup \sqcap_{F_2R_2})$, with $i \in \{1, 2\}$.

4.3 Coordination

Let $E = E_{R_1} \cup E_{R_2} \cup E_{B_1} \cup E_{B_2} \cup E_{F_1} \cup E_{F_2}$ be the set of events for the composite system in Equation 1. We formulate the coordination as described in Section 2 in terms of a satisfaction problem involving a safety property on TESs and a behavior property on the composite system. We first consider two safety properties:

$$P_{energy} = \{\sigma \in TES(E) \mid \forall i \in \mathbb{N}.\{(read(B_1), 0\%), (read(B_2), 0\%)\} \cap \mathsf{pr}_1(\sigma)(i) = \emptyset\}$$

$$P_{no-overlap} = \{\sigma \in TES(E) \mid \forall i \in \mathbb{N}.\forall l \in [0, 20]^2, \{(loc(I_1), l), (loc(I_2), l)\} \nsubseteq \mathsf{pr}_1(\sigma)(i)\}$$

The property P_{energy} collects all behaviors that never observe a battery value of 0%. The property $P_{no-overlap}$ describes all behaviors where the two robots are never observed together at the same location. Observe that, while both P_{energy} and $P_{no-overlap}$ specify some safety properties, they are not sufficient to ensure the safety of the system. We illustrate some scenarios with the property P_{energy}. If a component never reads its battery level, then the property P_{energy} is trivially satisfied, although effectively the battery may run out of energy. Also, if a component reads its battery level periodically, each of its readings may return an observation agreeing with the property. However, in between two read events, the battery may run out of energy (and somehow recharge). To circumvent those unsafe scenarios, we add an additional hyper-property.

Let $X_{read} = \{(read(B_1); l), (read(B_2); l) \mid l \in [0, 20]^2\}$ be the set of reading events for battery components B_1 and B_2. The property $\phi_{insert}(X_{read}, E)$, as detailed in Example 6, defines a class of component behaviors that are closed under insertion of *read* events for the battery component. Therefore, the system denoted as C, defined in Equation 1 is energy safe if $C \models P_{energy}$ and its behavior is closed under insertion of battery read events, i.e., $C \Vdash \phi_{insert}(X_{read}, E)$. In that case, every run of the system is part of a set that is closed under insertion, which means all read events that the robot may do in between two events observe a battery level greater than 0%. The behavior property enforces the following safety principle: had there been a violating behavior (i.e., a run where the battery has no energy), then an underlying TES would have observed it, and hence the hyperproperty would have been violated.

5 Related and future work

Our work offers a component-based semantics for cyber-physical systems [12, 11]. In [2], a similar aim is pursued by defining an algebra of components using

interface theory. Our component-based approach is inspired by [4, 5], where a component exhibits its behavior as a set of infinite timed-data streams. More details about co-algebraic techniques to prove component equivalences can be found in [16].

In [8], the authors describe an algebra of timed machines and of networks of timed machines. A timed machine is a state based description of a set of timed traces, such that every observation has a time stamp that is a multiple of a time step δ. The work differs from our current development in several respects. We focus in this paper on different algebraic operations on sets of timed-traces (TESs), and abstract away any underlying operational model (e.g., timed-automata). In [8], the authors explain how algebraic operations on timed machines *approximate* the intersection of sets of timed-traces. In our case, interaction is not restricted to input/output composition, but depends on the choice of a composability constraint on TESs and a composition function on observables. The work in [8] denotes an interesting class of components (closed under insertion of silent observation - r-closed) that deserves investigation.

Cyber-physical systems have also been studied from an actor perspective, where actors interact through events [18]. Problems of building synchronous protocols on top of asynchronous means of interaction are presented in [17].

Recent work has shown plenty of interest in studying the satisfaction problem of hyper-properties and the synthesis of reactive systems [9]. Some works focus more particularly on using hyper-properties for cyber-physical design [14].

The extension of hybrid automata [13] into a quantized hybrid automata is presented in [15], where the authors apply their model to give a formal semantics for data flow models of cyber-physical systems such as Simulink.

Compared to formalisms that model cyber-physical systems as more concrete operational or state-based mechanisms, such as automata or abstract machines, our more general abstract formalism is based only on the observable behavior of cyber-physical components and their composition into systems, regardless of what more concrete models or mechanisms may produce such behavior.

For future work, we want to provide a finite description for components, and use our current formalism as its formal semantics. In fact, we first started to model interactive cyber-physical systems as a set of finite state automata in composition, but the underlying complexity of automata interaction led us to introduce a more abstract component model to clarify the semantics of those interactions. Moreover, we want to investigate several proof techniques to show equivalences of components. We expect to be able to reason about local and global coordination, by studying how coordinators distribute over our different composition operators. Finally, our current work serves as a basis for defining a compositional semantics for a state-based component framework [1] written in Maude [7], a programming language based on rewriting logic. We will focus on evaluating the robustness of a set of components with respect to system requirements expressed as trace or hyper-properties. The complexity of the satisfaction problem requires some run-time techniques to detect deviations and produce meaningful diagnosis [10], a topic that we are currently exploring.

6 Conclusion

This paper contains three main contributions. First, we introduce a component model for cyber-physical systems where cyber and physical processes are uniformly described in terms of sequences of observations. Second, we provide ways to express interaction among components using algebraic operations, such as a parametric product and division, and give conditions under which product is associative, commutative, or idempotent. Third, we provide a formal basis to study trace and hyper-properties of components, and demonstrate the application of our work in an example describing several coordination problems.

Acknowledgement. Talcott was partially supported by the U. S. Office of Naval Research under award numbers N00014-15-1-2202 and N00014-20-1-2644, and NRL grant N0017317-1-G002.

References

[1] `https://scm.cwi.nl/FM/cp-agent`.

[2] Luca de Alfaro and Thomas A. Henzinger. "Interface Theories for Component-Based Design". In: *Embedded Software*. Ed. by Thomas A. Henzinger and Christoph M. Kirsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 148–165. ISBN: 978-3-540-45449-6.

[3] Bowen Alpern and Fred B. Schneider. "Defining liveness". In: *Information Processing Letters* 21.4 (1985), pp. 181–185. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/0020-0190(85)90056-0`.

[4] F. Arbab and J. J. M. M. Rutten. "A Coinductive Calculus of Component Connectors". In: *Recent Trends in Algebraic Development Techniques*. Ed. by Martin Wirsing, Dirk Pattinson, and Rolf Hennicker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 34–55. ISBN: 978-3-540-40020-2.

[5] Farhad Arbab. "Abstract Behavior Types: a foundation model for components and their composition". In: *Science of Computer Programming* 55.1 (2005). Formal Methods for Components and Objects: Pragmatic aspects and applications, pp. 3–52. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2004.05.010`.

[6] Michael R. Clarkson and Fred B. Schneider. "Hyperproperties". In: *J. Comput. Secur.* 18.6 (Sept. 2010), pp. 1157–1210. ISSN: 0926-227X.

[7] Manuel Clavel et al. *All About Maude: A High-Performance Logical Framework*. Springer, 2007.

[8] José Fiadeiro et al. "Dynamic networks of heterogeneous timed machines". In: *Mathematical Structures in Computer Science* 28.6 (2018), pp. 800–855. DOI: `10.1017/S0960129517000135`.

[9] Bernd Finkbeiner et al. "Realizing ω-regular Hyperproperties". In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 40–63. ISBN: 978-3-030-53291-8.

[10] Tobias Kappé et al. "Soft component automata: Composition, compilation, logic, and verification". In: *Science of Computer Programming* 183 (2019), p. 102300. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2019.08.001.

[11] K. Kim and P. R. Kumar. "Cyber–Physical Systems: A Perspective at the Centennial". In: *Proceedings of the IEEE* 100.Special Centennial Issue (2012), pp. 1287–1308. DOI: 10.1109/JPROC.2012.2189792.

[12] E. A. Lee. "Cyber Physical Systems: Design Challenges". In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008, pp. 363–369. DOI: 10.1109/ISORC.2008.25.

[13] Nancy Lynch, Roberto Segala, and Frits Vaandrager. "Hybrid I/O automata". In: *Information and Computation* 185.1 (2003), pp. 105–157. ISSN: 0890-5401. DOI: https://doi.org/10.1016/S0890-5401(03)00067-1.

[14] Luan Viet Nguyen et al. "Hyperproperties of Real-Valued Signals". In: MEMOCODE '17 (2017), pp. 104–113. DOI: 10.1145/3127041.3127058.

[15] Jin Woo Ro, Avinash Malik, and Partha Roop. "A Compositional Semantics of Simulink/Stateflow Based on Quantized State Hybrid Automata". In: *Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE '19. La Jolla, California: Association for Computing Machinery, 2019. ISBN: 9781450369978. DOI: 10.1145/3359986.3361198.

[16] J.J.M.M. Rutten. "Universal coalgebra: a theory of systems". In: *Theoretical Computer Science* 249.1 (2000). Modern Algebra, pp. 3–80. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(00)00056-6.

[17] Lui Sha et al. "PALS: Physically Asynchronous Logically Synchronous Systems". In: (June 2010).

[18] Carolyn Talcott. "Cyber-Physical Systems and Events". In: *Software Intensive Systems and New Computing Paradigms: Challenges and Visions*. Ed. by Martin Wirsing et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 101–115. ISBN: 978-3-540-89437-7. DOI: 10.1007/978-3-540-89437-7_6.

[19] Carolyn Talcott, Farhad Arbab, and Maneesh Yadav. "Soft Agents: Exploring Soft Constraints to Model Robust Adaptive Distributed Cyber-Physical Agent Systems". In: *Software, Services, and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*. Ed. by Rocco De Nicola and Rolf Hennicker. Cham: Springer International Publishing, 2015, pp. 273–290. ISBN: 978-3-319-15545-6. DOI: 10.1007/978-3-319-15545-6_18.

7 Appendix

7.1 Examples

Example 8. Consider a set of two events $E = \{0, 1\}$, and restrict our observations of E to $\{1\}$ and $\{0\}$. A component whose behavior contains TESs with alternating observations of $\{1\}$ and $\{0\}$ is defined by the tuple (E, L) where

$$L = \{\sigma \in TES(E) \mid \forall i \in \mathbb{N}.\ (\mathsf{pr}_1(\sigma)(i) = \{0\} \implies \mathsf{pr}_1(\sigma)(i+1) = \{1\}) \wedge (\mathsf{pr}_1(\sigma)(i) = \{1\} \implies \mathsf{pr}_1(\sigma)(i+1) = \{0\})\}$$

Note that this component is oblivious to time, and any stream of monotonically increasing non-Zeno real numbers would serve as a valid stream of time stamps for any such sequence of observations. ■

Example 9. We distinguish the usual *safety* and *liveness* properties [3, 6], and recall that every property can be written as the intersection of a safety and a liveness property. Let X be an arbitrary set, and P be a subset of $\mathbb{N} \to X$. Intuitively, P is safe if every *bad* stream not in P has a finite prefix every completion of which is bad, hence not in P. A property P is a liveness property if every finite sequence in X^* can be completed to yield an infinite sequence in P.
For instance, the property of terminating behavior for a component with interface E is a liveness property, defined as:

$$P_{\mathit{finite}}(E) = \{\sigma \in TES(E) \mid \exists n \in \mathbb{N}. \forall i > n.\ \mathsf{pr}_1(\sigma)(i) = \emptyset\}$$

$P_{\mathit{finite}}(E)$ says that, for every finite prefix of any stream in $TES(E)$, there exists a completion of that prefix with an infinite sequence of silent observations $\emptyset$ in $P_{\mathit{finite}}(E)$. ■

The next two examples present with more details the constraints imposed by the battery and the field components introduced in Section 4.

Example 10 (Field). A detailed example for the constraint P_F would be such that, for each sequence of observations, the output value of the *read* event corresponds to the current position of the robot given its previous moves. We will use a function called *pos* to determine the position of the robot after a sequence of observations. Let $s = \langle s_0, s' \rangle \in (\mathcal{P}(\mathbb{E}) \times \mathbb{R}_+)^*$ be a finite sequence observations. The position of the object I after a sequence of events is given by $pos(\langle s_0, s' \rangle) = pos(s_0) + pos(s')$ where, for $s_0 = (O, t)$:

$$pos((O,t)) = \begin{cases} (0, \eta) & if\ (move(I); (\mathsf{N}, \eta)) \in O \\ (0, -\eta) & if\ (move(I); (\mathsf{S}, \eta)) \in O \\ (\eta, 0) & if\ (move(I); (\mathsf{E}, \eta)) \in O \\ (-\eta, 0) & if\ (move(I); (\mathsf{W}, \eta)) \in O \\ (0, 0) & otherwise \end{cases}$$

$P_F(\sigma)$ is defined to accept all TESs such that all *read* events return the current position of the robot on the field, in accordance with the position function. Given $\sigma \in TES(E_F)$, $P_F(\sigma)$ is true if and only if

$$\forall i \in \mathbb{N}.\ (loc(I); p) \in \mathsf{pr}_1(\sigma)(i) \implies p = |pos(\sigma(0)....\sigma(i))|_{[-20,20]}$$

with $|(x,y)|_{[-20,20]} = (\min(\max(x,-20),20), \min(\max(y,-20),20))$.

P_F models the case where the robot starts in position $(0,0)$ and every move event changes the location of the robot on the field. Note that the move event has no effect if it occurs while the position is on the boundary of the field: it simulates the case of a fence, where moving against the fence would have the same observable as not moving.

■

Example 11 (Battery). An example for the structural constraints P_B is such that every *read* event instance returns the battery level as a function of the number of prior *move* events. We introduce the *lev* function, that takes a sequence of observations $s \in (\mathcal{P}(\mathbb{E}) \times \mathbb{R}_+)^*$ and returns the current charging level: $lev(s) = lev(\langle s(0), s(1)\rangle) + lev(\langle s(1)...s(i)\rangle)$, where

$$lev(\langle (O_1,t_1),(O_2,t_2)\rangle) = \begin{cases} \eta_1 + (t_2 - t_1)\eta_I & if\ discharge(B;\eta_1) \in O_1 \\ (t_2 - t_1)\eta_I & otherwise \end{cases}$$

where η_I triggers a discharge linearly proportional to the time spent between two observations, and η_1 is a discharge coefficient proportional to the speed of the robot with whom the battery interact (see Section 4). The battery predicate $P_{battery}$ is defined such that all $read(B;s)$ events return the current battery level of the robot, in accordance with the *lev* function, i.e., for all $\sigma \in TES(\mathbb{E})$:

$$\forall i \in \mathbb{N}.\mathsf{pr}_1(\sigma)(i) = read(B;c) \implies c = \max(100 - lev(\langle \sigma(0),...,\sigma(i)\rangle), 0)$$

The constraint P_B models the case where the battery initially starts at an energy level of 100, decreases after each move event, and has a linear discharge factor over time. Remark that different predicates P_B account for different model of batteries. Alternatively, the discharge factor could depend on external parameters (temperature, discharge level, etc), adding a non-linear side to the model. Our model allows for such specification, where the function *lev* takes an additional sequence of parameters in $\mathbb{R}_+$, and where discharge coefficients are parametric, i.e., $\eta_1, \eta_I : \mathbb{R}_+ \to [0,1]$. Then, given a finite sequence of observations $s \in (\mathcal{P}(\mathbb{E}) \times \mathbb{R}_+)^*$ and a finite sequence of parameters $p \in \mathbb{R}_+^*$, we extend *lev* to the function lev^*: $lev^*(s,p) = lev^*(\langle s(0), s(1)\rangle, p(0)) + lev^*(\langle s(1)...s(i)\rangle, \langle p(1),....,p(i)\rangle)$, where

$$lev^*(\langle (O_1,t_1),(O_2,t_2)\rangle, p) = \begin{cases} \eta_1(p) + (t_2 - t_1)\eta_I(p) & if\ discharge(B;\eta_1) \in O_1 \\ (t_2 - t_1)\eta_I(p) & otherwise \end{cases}$$

■

7.2 Lemma

The following lemma gives some sufficient condition on a composability relation on observations to lift to an associative composability relation on TESs. Intuitively, those conditions stipulate that the composability relation is transitive, and satisfies other closure conditions. We use Lemma 7 to prove associativity of $\bowtie_\sqcap$ in Corollary 2.

Lemma 7. *Let $+$ be an associative composition function, and let κ be a composability relation on observations such that, for any $a = (o_1, t_1) \in \mathcal{P}(E_1) \times \mathbb{R}_+$, $b = (o_2, t_2) \in \mathcal{P}(E_2) \times \mathbb{R}_+$, and $c(o_3, t_3) \in \mathcal{P}(E_3) \times \mathbb{R}_+$:*

1. *if $t_1 < t_2$, then*
   $((a, b) \in \kappa(E_1, E_2) \wedge (a, d) \in \kappa(E_1, E_3)) \implies (b, d) \in \kappa(E_2, E_3)$
2. *if $t_3 < t_2$, then*
   $((a, d) \in \kappa(E_1, E_3) \wedge (b, d) \in \kappa(E_2, E_3)) \implies (a, b) \in \kappa(E_1, E_2)$
3. *if $t_2 < t_1$ or $t_2 < t_3$, then*
   $((a, b) \in \kappa(E_1, E_2) \wedge (b, d) \in \kappa(E_2, E_3)) \implies (a, d) \in \kappa(E_1, E_3)$
4. *if $t_2 < t_3$, then*
   $(a, b) \in \kappa(E_1, E_2) \iff (a, b) \in \kappa(E_1, E_2 \cup E_3)$
5. *if $t_1 < t_2$ or $t_3 < t_2$, then*
   $(a, d) \in \kappa(E_1, E_3) \iff ((a, d) \in \kappa(E_1 \cup E_2, E_3) \wedge (a, d) \in \kappa(E_1, E_2 \cup E_3))$
6. *if $t_2 < t_1$, then*
   $(b, d) \in \kappa(E_2, E_3) \iff (b, d) \in \kappa(E_1 \cup E_2, E_3)$
7. *if $t_1 = t_2$, then*
   $((o_1 + o_2, t), c) \in \kappa(E_1 \cup E_2, E_3) \iff ((o_1, t), c) \in \kappa(E_1, E_3) \wedge ((o_2, t), c) \in \kappa(E_2, E_3))$
8. *if $t_2 = t_3$, then*
   $(a, (o_2 + o_3, t)) \in \kappa(E_1, E_2 \cup E_3) \iff (a, (o_2, t)) \in \kappa(E_1, E_2) \wedge (a, (o_3, t)) \in \kappa(E_1, E_3))$

Then, $\times_{([\kappa],[+])}$ is associative.

Corollary 2. *We fix the set union $\cup$ as composition function on observables. Let $\sqcap \subseteq \mathcal{P}(\mathbb{E}) \times \mathcal{P}(\mathbb{E})$ be a relation on observables. Let $\bowtie_\sqcap$ be the composability relation defined in Definition 11, and let $\kappa'_{sync,\sqcap}$ be such that, for any $(O_1, t_1) \in \mathcal{P}(E_1) \times \mathbb{R}_+$ and $(O_2, t_2) \in \mathcal{P}(E_2) \times \mathbb{R}_+$:*

$$((O_1, t_1), (O_2, t_2)) \in \kappa'_{sync,\sqcap}(E_1, E_2) \iff (t_1 = t_2 \wedge ((O_1, t_1), (O_2, t_2)) \in \kappa_{sync,\sqcap}(E_1, E_2))$$

We write $\bowtie'_\sqcap$ for the lifting of $\kappa'_{sync,\sqcap}$. Then:

- *$\times_{(\bowtie_\sqcap,[\cup])}$ is commutative if $\sqcap$ is symmetric,*
- *$\times_{(\bowtie'_\sqcap,[\cup])}$ is associative if $\sqcap$ be the smallest relation such that, for all O_1, O_2, and $O_3 \in \mathbb{E}$, $(O_1 \cup O_2, O_3) \in \sqcap \iff ((O_1, O_3) \in \sqcap \wedge (O_2, O_3) \in \sqcap)$ and $(O_1, O_2 \cup O_3) \in \sqcap \iff ((O_1, O_3) \in \sqcap \wedge (O_2, O_3) \in \sqcap)$.*
- *$\times_{(\bowtie_\sqcap,[\cup])}$ is idempotent if $\sqcap$ is the smallest relation such that, for all non empty $O \subseteq \mathbb{E}$, $(O, O) \in \sqcap$.*

7.3 Proofs

Proof (Lemma 1). Commutativity. Let $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$ be two components, and $(R, \oplus)$ be a pair of a composability relation and composition function on TESs. We write $C = (E, L) = C_1 \times_{(R,\oplus)} C_2$ and $C' = (E', L') = C_2 \times_{(R,\oplus)} C_1$. We first observe that $E = E_1 \cup E_2 = E'$. The condition for the product of two components to be commutative reduces to showing that $L = L'$, also equivalently written as:

$$\begin{aligned} L = L' \iff & \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ (\sigma_1, \sigma_2) \in R(E_1, E_2)\} \\ & = \{\sigma_2 \oplus \sigma_1 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ (\sigma_2, \sigma_1) \in R(E_2, E_1)\} \end{aligned}$$

If R is symmetric (as in Definition 3) and $\oplus$ is commutative, then $L = L'$. Hence, if R is symmetric and $\oplus$ is commutative, then $\times_{(R,\oplus)}$ is commutative.

Oppositely, if R is symmetric and $L = L'$, we show that $\oplus$ is commutative. We take the symmetric relation $R(E_1, E_2)$ such that $(\sigma_1, \sigma_2) \in R(E_1, E_2)$ for any $\sigma_1 \in TES(E_1)$ and $\sigma_2 \in TES(E_2)$. Let C_σ be the component $(E_\sigma, \{\sigma\})$ where $E_\sigma = \bigcup\{\sigma(i) \mid i \in \mathbb{N}\}$. Thus, for any $\sigma_1 \in TES(E_1)$ and $\sigma_2 \in TES(E_2)$, $C_{\sigma_1} \times_{(R,\oplus)} C_{\sigma_2} = (E_{\sigma_1} \cup E_{\sigma_2}, \{\sigma_1 \oplus \sigma_2\}$. A necessary condition for $\times_{(R,\oplus)}$ to be commutative is that $\{\sigma_1 \oplus \sigma_2\} = \{\sigma_2 \oplus \sigma_1\}$, which imposes commutativity on $\oplus$.

Associativity. Let $(R_1, \oplus_1)$ and $(R_2, \oplus_2)$ be two pairs of a composability relation and a composition function on TESs. We consider three components $C_i = (L_i, E_i)$, with $i \in \{1, 2, 3\}$.

The set of events for component $((C_1 \times_{(R_1,\oplus_1)} C_2) \times_{(R_2,\oplus_2)} C_3)$ is the set $E_1 \cup E_2 \cup E_3$, which is equal to the set of events for component $(C_1 \times_{(R_1,\oplus_1)} (C_2 \times_{(R_2,\oplus_2)} C_3))$.

Let L' and L'' respectively be the behaviors of components $(C_1 \times_{(R_1,\oplus_1)} C_2) \times_{(R_2,\oplus_2)} C_3$ and $C_1 \times_{(R_1,\oplus_1)} (C_2 \times_{(R_2,\oplus_2)} C_3)$. We show some sufficient conditions for $L' = L''$, also written as

$$\begin{aligned} & \{(\sigma_1 \oplus_1 \sigma_2) \oplus_2 \sigma_3 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ \sigma_3 \in L_3.\ (\sigma_1, \sigma_2) \in R_1(E_1, E_2) \wedge \\ & \qquad (\sigma_1 \oplus_1 \sigma_2, \sigma_3) \in R_2(E_1 \cup E_2, E_3)\} \\ & = \{\sigma_1 \oplus_1 (\sigma_2 \oplus_2 \sigma_3) \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2,\ \sigma_3 \in L_3.\ (\sigma_2, \sigma_3) \in R_2(E_2, E_3) \wedge \\ & \qquad (\sigma_1, \sigma_2 \oplus_2 \sigma_3) \in R_1(E_1, E_2 \cup E_3)\} \end{aligned}$$

We first observe that if $\sigma_1 \oplus_1 (\sigma_2 \oplus_2 \sigma_3) = (\sigma_1 \oplus_1 \sigma_2) \oplus_2 \sigma_3$ then a sufficient condition for L' to be equal to L'' is that

$$\begin{aligned} & (\sigma_1, \sigma_2) \in R_1(E_1, E_2) \wedge (\sigma_1 \oplus_1 \sigma_2, \sigma_3) \in R_2(E_1 \cup E_2, E_3) \iff \\ & (\sigma_2, \sigma_3) \in R_2(E_2, E_3) \wedge (\sigma_1, \sigma_2 \oplus_2 \sigma_3) \in R_1(E_1, E_2 \cup E_3) \end{aligned}$$

for every $(\sigma_1, \sigma_2, \sigma_3) \in L_1 \times L_2 \times L_3$. Assuming that R_1 and R_2 satisfies the following constraint for every $(\sigma_1, \sigma_2, \sigma_3) \in L_1 \times L_2 \times L_3$:

$$\begin{aligned} & (\sigma_1, \sigma_2) \in R_1(E_1, E_2) \wedge (\sigma_1 \oplus_1 \sigma_2, \sigma_3) \in R_2(E_1 \cup E_2, E_3) \iff \\ & (\sigma_2, \sigma_3) \in R_2(E_2, E_3) \wedge (\sigma_1, \sigma_2 \oplus_2 \sigma_3) \in R_1(E_1, E_2 \cup E_3) \end{aligned}$$

we show that $\times_{(R_1,\oplus_1)}$ and $\times_{(R_2,\oplus_2)}$ are associative if and only if $\sigma_1 \oplus_1 (\sigma_2 \oplus_2 \sigma_3) = (\sigma_1 \oplus_1 \sigma_2) \oplus_2 \sigma_3$ for every $(\sigma_1, \sigma_2, \sigma_3) \in L_1 \times L_2 \times L_3$. The proof is similar to the case of commutativity, i.e. showing that $C_{\sigma_1} \times_{(R_1,\oplus_1)} (C_{\sigma_2} \times_{(R_2,\oplus_2)} C_{\sigma_3}) = (C_{\sigma_1} \times_{(R_1,\oplus_1)} C_{\sigma_2}) \times_{(R_2,\oplus_2)} C_{\sigma_3}$ implies that $\sigma_1 \oplus_1 (\sigma_2 \oplus_2 \sigma_3) = (\sigma_1 \oplus_1 \sigma_2) \oplus_2 \sigma_3$. Thus, if R_1 and R_2 satisfies the constraint as written above, $\times_{(R_1,\oplus_1)}$ and $\times_{(R_2,\oplus_2)}$ are associative if and only if $\oplus_1$ and $\oplus_2$ are associative.

Idempotency. We show that if for all $E_1 \subseteq \mathbb{E}$, and $\sigma, \tau \in TES(E_1)$, we have that $\overline{(\sigma,\tau) \in R_1(E_1,E_1)} \implies \sigma = \tau$, then $\times_{(R_1,\oplus_1)}$ is idempotent if and only if $\oplus_1$ is idempotent. We first observe that, given a component $C = (E, L)$, the component $C \times_{(R_1,\oplus_1)} C = (E, L')$ has the same set of events, E.

We show that $(\sigma_1, \sigma_2) \in R(E_1, E_1) \implies \sigma_1 = \sigma_2$ and $\oplus_1$ idempotent is a sufficient condition for having $L' = L$. Indeed,

$$\begin{aligned} L' &= \{\sigma_1 \oplus_1 \sigma_2 \mid \sigma_1, \sigma_2 \in L,\ (\sigma_1, \sigma_2) \in R_1(E_1, E_1)\} \\ &= \{\sigma_1 \oplus_1 \sigma_1 \mid \sigma_1 \in L\} \\ &= L_1 \end{aligned}$$

Similar to the previous cases, if for all $E_1 \subseteq \mathbb{E}$, and $\sigma, \tau \in TES(E_1)$, we have that $(\sigma,\tau) \in R_1(E_1,E_1) \implies \sigma = \tau$ then $\times_{(R,\oplus_1)}$ is idempotent if and only if $\oplus_1$ is idempotent. The proof is similar to the case of commutativity, i.e. showing that $C_{\sigma_1} \times_{(R_1,\oplus_1)} C_{\sigma_1} = C_{\sigma_1}$ implies that $\sigma_1 \oplus_1 \sigma_1 = \sigma_1$.

□

Proof (Lemma 2). Let $(R, \oplus)$ be a pair of a composability relation and a composition function on TESs. For any components $C_1 = (E_1, L_1)$ and $C_2 = (E_2, L_2)$, let $(C_1 \times_{(R,\oplus)} C_2) = (E_1 \cup E_2, L')$ and $(C_1 \times_{(R,\oplus)} C_2)/_{(R,\oplus)} C_2 = (E, L)$. The set L is such that, for any $\sigma \in TES(E_1 \cup E_2)$:

$$\sigma \in L \iff \exists \sigma_2 \in L_2, \sigma' \in L'.\ (\sigma, \sigma_2) \in R(E_1 \cup E_2, E_2) \wedge \sigma' = (\sigma \oplus \sigma_2)$$

By construction, the existence of $\sigma' \in L'$ is equivalent to the existence of $\sigma'_1 \in L_1$ and $\sigma'_2 \in L_2$ such that

$$(\sigma'_1, \sigma'_2) \in R(E_1, E_2) \wedge \sigma' = (\sigma'_1 \oplus \sigma'_2)$$

Thus, for any $\sigma \in TES(E_1 \cup E_2)$:

$$\begin{aligned} \sigma \in L \iff & \exists \sigma'_1 \in L_1, \sigma'_2 \in L_2, \sigma_2 \in L_2. \\ & (\sigma, \sigma_2) \in R(E_1 \cup E_2, E_2) \wedge \sigma \oplus \sigma_2 = \sigma'_1 \oplus \sigma'_2 \wedge (\sigma'_1, \sigma'_2) \in R(E_1, E_2) \\ \impliedby & \sigma \in L_1 \wedge \exists \sigma_2 \in L_2.(\sigma, \sigma_2) \in R(E_1 \cup E_2, E_2) \wedge (\sigma, \sigma_2) \in R(E_1, E_2) \end{aligned}$$

The last implication concludes the proof that

$$\{\sigma \in L_1 \mid \exists \sigma_2 \in L_2.(\sigma, \sigma_2) \in R(E_1 \cup E_2, E_2) \wedge (\sigma, \sigma_2) \in R(E_1, E_2)\} \subseteq L$$

□

Proof (Lemma 3). Let κ be a composability relation on observations, and let $E_1, E_2 \subseteq \mathbb{E}$. We recall that $\Phi_\kappa(E_1, E_2)$ is such that, for any $\mathcal{R} \subseteq TES(\mathbb{E})^2$:

$$\begin{aligned}\Phi_\kappa(E_1,E_2)(\mathcal{R}) = \{(\tau_1,\tau_2) \mid\ & (\tau_1(0),\tau_2(0)) \in \kappa(E_1,E_2) \wedge \\ & (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ & (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R} \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R} \vee \\ & \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R})\}\end{aligned}$$

Let $\mathcal{R}_1, \mathcal{R}_2 \subseteq TES(E)^2$ such that $\mathcal{R}_1 \subseteq \mathcal{R}_2$. We show that $\Phi_\kappa(E_1,E_2)(\mathcal{R}_1) \subseteq \Phi_\kappa(E_1,E_2)(\mathcal{R}_2)$. For any $(\tau_1,\tau_2) \in TES(E)^2$,

$$\begin{aligned}(\tau_1,\tau_2) \in \Phi_\kappa(E_1,E_2)(\mathcal{R}_1) \iff & (\tau_1(0),\tau_2(0)) \in \kappa(E_1,E_2) \wedge \\ & (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ & (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R}_1 \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R}_1 \vee \\ & \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R}_1) \\ \implies & (\tau_1(0),\tau_2(0)) \in \kappa(E_1,E_2) \wedge \\ & (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ & (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R}_2 \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R}_2 \vee \\ & \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R}_2) \\ \implies & (\tau_1,\tau_2) \in \Phi_\kappa(E_1,E_2)(\mathcal{R}_2)\end{aligned}$$

Therefore, $\mathcal{R}_1 \subseteq \mathcal{R}_2$ implies that $\Phi_\kappa(E_1,E_2)(\mathcal{R}_1) \subseteq \Phi_\kappa(E_1,E_2)(\mathcal{R}_2)$, and we conclude that $\Phi_\kappa(E_1,E_2)$ is monotonic. By the greatest fixed point theorem, $\Phi_\kappa(E_1,E_2)$ has a greatest fixed point $P = \Phi_\kappa(E_1,E_2)(P)$ such that:

$$P = \bigcup \{\mathcal{R} \mid \mathcal{R} \subseteq \Phi_\kappa(E_1,E_2)(\mathcal{R})\}$$

□

Proof (Lemma 4). We first note that, given a composability relation κ on observables, the lifting $[\kappa]$ is a composability relation on TESs. Indeed, for any pair of interfaces $E_1, E_2 \subseteq \mathbb{E}$, any $(\sigma,\tau) \in [\kappa](E_1,E_2)$ is a pair in $TES(E_1) \times TES(E_2)$.

If κ is symmetric (as in Definition 3), we show that $[\kappa]$ is also symmetric. Given a set $\mathcal{R} \subseteq TES(\mathbb{E}) \times TES(\mathbb{E})$, we use the notation $\overline{\mathcal{R}}$ to denote the smallest set such that $(\sigma,\tau) \in \mathcal{R} \iff (\tau,\sigma) \in \overline{\mathcal{R}}$.

If κ is symmetric, then for $\mathcal{R} \subseteq TES(\mathbb{E}) \times TES(\mathbb{E})$,

$$\begin{aligned}\Phi_\kappa(E_1,E_2)(\mathcal{R}) &= \{(\tau_1,\tau_2) \mid (\tau_1(0),\tau_2(0)) \in \kappa(E_1,E_2) \wedge \\ &\qquad (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ &\qquad (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R} \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R} \vee \\ &\qquad \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R})\} \\ &= \{(\tau_1,\tau_2) \mid (\tau_2(0),\tau_1(0)) \in \kappa(E_2,E_1) \wedge \\ &\qquad (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ &\qquad (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R} \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R} \vee \\ &\qquad \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R})\} \\ &= \{(\tau_1,\tau_2) \mid (\tau_2(0),\tau_1(0)) \in \kappa(E_2,E_1) \wedge \\ &\qquad (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ &\qquad (t_1 < t_2 \wedge (\tau_2, \tau_1') \in \overline{\mathcal{R}} \vee t_2 < t_1 \wedge (\tau_2', \tau_1) \in \overline{\mathcal{R}} \vee \\ &\qquad \ t_2 = t_1 \wedge (\tau_2', \tau_1') \in \overline{\mathcal{R}})\} \\ &= \{(\tau_1,\tau_2) \mid (\tau_2,\tau_1) \in \Phi_\kappa(E_2,E_1)(\overline{\mathcal{R}})\} \qquad (1)\end{aligned}$$

which shows that $[\kappa]$ is symmetric since, for any $E_1, E_2 \subseteq \mathbb{E}$, $[\kappa](E_1, E_2) = \bigcup_{\mathcal{R} \subseteq TES(\mathbb{E}) \times TES(\mathbb{E})} \{\mathcal{R} \mid \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R})\}$, and

$$\begin{array}{rl} (\sigma, \tau) \in [\kappa](E_1, E_2) & \iff \exists \mathcal{R}.\ (\sigma, \tau) \in \mathcal{R} \wedge \mathcal{R} \subseteq \Phi_\kappa(E_1, E_2)(\mathcal{R}) \\ & \iff \exists \overline{\mathcal{R}}.\ (\tau, \sigma) \in \overline{\mathcal{R}} \wedge \overline{\mathcal{R}} \subseteq \Phi_\kappa(E_2, E_1)(\overline{\mathcal{R}}) \\ & \iff (\tau, \sigma) \in [\kappa](E_2, E_1) \end{array}$$

where the first equivalence is given by the fact that $[\kappa](E_1, E_2)$ is a post fixed point of $\Phi_\kappa(E_1, E_2)$, the second equivalence is obtained from equality (1), and the third equivalence is given by the fact that $[\kappa](E_2, E_1)$ is the greatest post fixed point.

□

Proof (Lemma 5). Commutativity. From Lemma 4, if κ is symmetric, then its lifting $[\kappa]$ is also symmetric. Therefore, it is sufficient for κ to be symmetric and for $+$ to be commutative in order for $[\kappa]$ to be symmetric and $[+]$ to be commutative, and therefore $\times_{([\kappa],[+])}$ to be commutative.

Associativity. A sufficient condition for the products $\times_{([\kappa_1],[+_1])}$ and $\times_{([\kappa_2],[+_2])}$ to be associative is that, for every $\sigma_i \in TES(E_i)$ for $i \in \{1, 2, 3\}$, we have that $\sigma_1[+_1](\sigma_2[+_2]\sigma_3) = \sigma_1[+_1](\sigma_2[+_2]\sigma_3)$ and

$$\begin{array}{c} P_1 := (\sigma_1, \sigma_2) \in [\kappa_1](E_1, E_2) \wedge (\sigma_1[+_1]\sigma_2, \sigma_3) \in [\kappa_2](E_1 \cup E_2, E_3) \iff \\ (\sigma_2, \sigma_3) \in [\kappa_2](E_2, E_3) \wedge (\sigma_1, \sigma_2[+_2]\sigma_3) \in [\kappa_1](E_1, E_2 \cup E_3) \end{array}$$

Let κ_1 and κ_2 be such that, for any $o_i \in (\mathcal{P}(E_i) \times \mathbb{R}_+)$ with $i \in \{1, 2, 3\}$ we have

$$\begin{array}{c} P_2 := ((o_1, o_2) \in \kappa(E_1, E_2) \wedge (\iota_1(o_1, o_2), o_3) \in \kappa(E_1 \cup E_2, E_3)) \iff \\ ((o_2, o_3) \in \kappa(E_2, E_3) \wedge (o_1, \iota_2(o_2, o_3)) \in \kappa(E_1, E_2 \cup E_3)) \end{array}$$

with $k \in \{1, 2\}$ and $\iota_k((O, t), (P, l)) = \begin{cases} (O, t) & \text{if } t < l \\ (P, l) & \text{if } l < t \\ (O +_k P, t) \end{cases}$

We show that $P_2 \implies P_1$. We introduce the function

$$\begin{array}{rl} \Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)(\mathcal{R}) = \{\ (\sigma_1, \sigma_2, \sigma_3) \mid & (\sigma_1(0), \sigma_2(0)) \in \kappa_1(E_1, E_2) \wedge \\ & ((\sigma_1[+_1]\sigma_2)(0), \sigma_3(0)) \in \kappa_2(E_1 \cup E_2, E_3) \wedge \\ & \mathsf{pr}_2(\sigma_1)(0) = t_1 \wedge \mathsf{pr}_2(\sigma_2)(0) = t_2 \wedge \mathsf{pr}_2(\sigma_3)(0) = t_3 \wedge \\ & (t_1 < t_2 \wedge t_1 < t_3 \wedge (\sigma_1', \sigma_2, \sigma_3) \in \mathcal{R} \vee \\ & \ t_2 < t_1 \wedge t_2 < t_3 \wedge (\sigma_1, \sigma_2', \sigma_3) \in \mathcal{R} \vee \\ & \ t_3 < t_2 \wedge t_3 < t_1 \wedge (\sigma_1, \sigma_2, \sigma_3') \in \mathcal{R} \vee \\ & \ t_1 = t_2 \wedge t_1 < t_3 \wedge (\sigma_1', \sigma_2', \sigma_3) \in \mathcal{R} \vee \\ & \ t_2 = t_3 \wedge t_2 < t_1 \wedge (\sigma_1, \sigma_2', \sigma_3') \in \mathcal{R} \vee \\ & \ t_1 = t_3 \wedge t_1 < t_2 \wedge (\sigma_1', \sigma_2, \sigma_3') \in \mathcal{R} \vee \\ & \ t_1 = t_3 \wedge t_1 = t_2 \wedge (\sigma_1', \sigma_2', \sigma_3') \in \mathcal{R})\} \end{array}$$

The function $\Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)$ is monotone, and we use Γ to denote its greatest post fixed point. Let $\Gamma_{1,2} = \{(\sigma_1, \sigma_2) \mid (\sigma_1, \sigma_2, \sigma_3) \in \Gamma\}$. Due to the construction

of $\Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)$, we have that $\Gamma_{1,2} \subseteq [\kappa_1](E_1, E_2)$. Similarly, let $\Gamma_{1+_1 2,3} = \{(\sigma_1[+_1]\sigma_2, \sigma_3) \mid (\sigma_1, \sigma_2, \sigma_3) \in \Gamma\}$, we have $\Gamma_{1+_1 2,3} \subseteq [\kappa_2](E_1 \cup E_2, E_3)$. Thus, for any triple $(\sigma_1, \sigma_2, \sigma_3) \in \Gamma$, the left hand side of predicate P_1 is true, i.e. $(\sigma_1, \sigma_2) \in [\kappa_1](E_1, E_2) \wedge (\sigma_1[+_1]\sigma_2, \sigma_3) \in [\kappa_2](E_1 \cup E_2, E_3)$ is true.

Alternatively, let

$$\Xi = \{(\sigma_1, \sigma_2, \sigma_3) \mid (\sigma_1, \sigma_2) \in [\kappa_1](E_1, E_2) \wedge (\sigma_1[+_1]\sigma_2, \sigma_3) \in [\kappa_2](E_1 \cup E_2, E_3)\}$$

We show that $\Xi \subseteq \Gamma$, by showing that $\Xi \subseteq \Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)(\Xi)$. By construction, any triple $(\sigma_1, \sigma_2, \sigma_3) \in \Xi$ satisfies the condition $(\sigma_1(0), \sigma_2(0)) \in \kappa_1(E_1, E_2) \wedge ((\sigma_1[+_1]\sigma_2)(0), \sigma_3(0)) \in \kappa_2(E_1 \cup E_2, E_3)$ of $\Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)$. Moreover, given $(\sigma_1, \sigma_2, \sigma_3) \in \Xi$, with $\mathsf{pr}_2(\sigma_1)(0) = t_1 \wedge \mathsf{pr}_2(\sigma_2)(0) = t_2 \wedge \mathsf{pr}_2(\sigma_3)(0) = t_3$, we split cases based on time and show that $\Xi \subseteq \Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)(\Xi)$. As an example, if $t_1 < t_2 \wedge t_1 < t_3$, we have that $(\sigma_1', \sigma_2) \in [\kappa_1](E_1, E_2) \wedge (\sigma_1'[+_1]\sigma_2, \sigma_3) \in [\kappa_2](E_1 \cup E_2, E_3)$ and therefore $(\sigma_1', \sigma_2, \sigma_3) \in \Xi$.

Therefore, $\Xi \subseteq \Psi_{\kappa_1,\kappa_2}(E_1, E_2, E_3)(\Xi)$, and $\Xi \subseteq \Gamma$. Then,

$$(\sigma_1, \sigma_2, \sigma_3) \in \Gamma \iff (\sigma_1, \sigma_2) \in [\kappa_1](E_1, E_2) \wedge (\sigma_1[+_1]\sigma_2, \sigma_3) \in [\kappa_2](E_1 \cup E_2, E_3)$$

In the case where P_2 is true, using a similar constuction, we can show that

$$(\sigma_1, \sigma_2, \sigma_3) \in \Gamma \iff (\sigma_2, \sigma_3) \in [\kappa_2](E_2, E_3) \wedge (\sigma_1, \sigma_2[+_2]\sigma_3) \in [\kappa_1](E_1, E_2 \cup E_3)$$

We conclude that if P_2 is true, then P_1 is true.

We show that if $(O_1 +_1 O_2) +_2 O_3 = O_1 +_1 (O_2 +_2 O_3)$ for every $O_i \in \mathcal{P}(E_i)$ with $i \in \{1, 2, 3\}$, then $\sigma_1[+_1](\sigma_2[+_2]\sigma_3) = (\sigma_1[+_1]\sigma_2)[+_2]\sigma_3$ for every $\sigma_i \in L_i$ with $i \in \{1, 2, 3\}$.

We split cases, and we write $\sigma_i(0) = (O_i, t_i)$:

$$\sigma_1[+_1](\sigma_2[+_2]\sigma_3) = \begin{cases} \langle (O_1, t_1) \rangle \cdot (\sigma_1'[+_1](\sigma_2[+_2]\sigma_3) & \text{if } t_1 < t_2 \wedge t_1 < t_3 \\ \langle (O_2, t_2) \rangle \cdot (\sigma_1[+_1](\sigma_2'[+_2]\sigma_3) & \text{if } t_2 < t_1 \wedge t_2 < t_3 \\ \langle (O_3, t_3) \rangle \cdot (\sigma_1[+_1](\sigma_2[+_2]\sigma_3') & \text{if } t_3 < t_2 \wedge t_3 < t_1 \\ \langle (O_1 +_1 O_2, t_1) \rangle \cdot (\sigma_1'[+_1](\sigma_2'[+_2]\sigma_3) & \text{if } t_1 = t_2 \wedge t_1 < t_3 \\ \langle (O_2 +_2 O_3, t_2) \rangle \cdot (\sigma_1[+_1](\sigma_2'[+_2]\sigma_3') & \text{if } t_2 = t_3 \wedge t_2 < t_1 \\ \langle (O_1 +_1 O_3, t_1) \rangle \cdot (\sigma_1'[+_1](\sigma_2[+_2]\sigma_3') & \text{if } t_1 = t_3 \wedge t_1 < t_2 \\ \langle (O_1 +_1 (O_2 +_2 O_3), t_1) \rangle \cdot (\sigma_1'[+_1](\sigma_2'[+_2]\sigma_3') & \text{if } t_1 = t_3 \wedge t_1 = t_2 \end{cases}$$

The only case that differs from $(\sigma_1[+_1]\sigma_2)[+_2]\sigma_3$ is when $t_1 = t_3 = t_2$, which gives $((O_1 +_1 O_2) +_2 O_3, t_1)$. Thus, if $((O_1 +_1 O_2) +_2 O_3, t_1) = (O_1 +_1 (O_2 +_2 O_3), t_1)$ for every $O_i \in \mathcal{P}(E_i)$ with $i \in \{1, 2, 3\}$, then $\sigma_1[+_1](\sigma_2[+_2]\sigma_3) = \sigma_1[+_1](\sigma_2[+_2]\sigma_3)$ for every $\sigma_i \in L_i$ with $i \in \{1, 2, 3\}$.

Idempotency. Let $+_1$ be idempotent, then the lifting $[+_1]$ is also idempotent.

We show that, for all $E \subseteq \mathbb{E}$ and $o_1, o_2 \in \mathcal{P}(E) \times \mathbb{R}_+$ we have $(o_1, o_2) \in \kappa_1(E, E) \implies o_1 = o_2$, then for all $\sigma, \tau \in TES(E)$, $(\sigma, \tau) \in [\kappa_1](E, E) \implies \sigma = \tau$, which is a sufficient condition for $\times_{([\kappa_1],[+_1])}$ to be idempotent.

By definition $[\kappa_1](E,E)$ is the greatest fixed point of the function:

$$\begin{aligned}\Phi_{\kappa_1}(E,E)(R) = \{(\tau_1,\tau_2) \mid\ & (\tau_1(0),\tau_2(0)) \in \kappa_1(E,E) \wedge \\ & (\mathsf{pr}_2(\tau_1)(0) = t_1 \wedge \mathsf{pr}_2(\tau_2)(0) = t_2) \wedge \\ & (t_1 < t_2 \wedge (\tau_1', \tau_2) \in \mathcal{R} \vee t_2 < t_1 \wedge (\tau_1, \tau_2') \in \mathcal{R} \\ & \ t_2 = t_1 \wedge (\tau_1', \tau_2') \in \mathcal{R})\} \\ \subseteq \{(\tau_1,\tau_2) \mid\ & \tau_1(0) = \tau_2(0) \wedge (\tau_1', \tau_2') \in \mathcal{R}\}\end{aligned}$$

Therefore, we conclude that $[\kappa_1](E,E) \subseteq \{(\sigma,\sigma) \mid \sigma \in TES(E)\}$.

That concludes the proof. □

Proof (Lemma 6). Since $C_P = (E, L_P)$ has the same interface as the component $C = (E, L)$, and given that $\sqcap$ satisfies the condition for using corollary 2, the product $\times_{(\bowtie_\sqcap, [\cup])}$ is idempotent. Let $C \times_{(\bowtie_\sqcap, \oplus)} C_P = (E, L')$. If $(\sigma, \tau) \in L'$ then $\sigma = \tau$. Thus, $L' \subseteq L \cap L_P$.

Alternatively, let $\sigma \in L \cap L_P$. We observe that at any point $n \in \mathbb{N}$, we have $(\sigma(n), \sigma(n)) \in \bowtie_\sqcap (E,E)$, since, given $\sigma(n) = (O_n, t_n)$ and the assumption on $\sqcap$, we have $(O_n, O_n) \in \sqcap$ or $O_n = \emptyset$. Therefore, $(\sigma,\sigma) \in \bowtie_\sqcap$.

We conclude that $L \cap L_P = L'$. □

Proof (Lemma 7). Let κ be a composability relation on observations and $+$ a composition function on observations. We state the following assumptions on κ, for any $a = (O_1, t_1) \in \mathcal{P}(E_1) \times \mathbb{R}_+$, $b = (O_2, t_2) \in \mathcal{P}(E_2) \times \mathbb{R}_+$, and $c = (O_3, t_3) \in \mathcal{P}(E_3) \times \mathbb{R}_+$:

1. if $t_1 < t_2$, then
   $((a,b) \in \kappa(E_1,E_2) \wedge (a,d) \in \kappa(E_1,E_3)) \implies (b,d) \in \kappa(E_2,E_3)$
2. if $t_3 < t_2$, then
   $((a,d) \in \kappa(E_1,E_3) \wedge (b,d) \in \kappa(E_2,E_3)) \implies (a,b) \in \kappa(E_1,E_2)$
3. if $t_2 < t_1$ or $t_2 < t_3$, then
   $((a,b) \in \kappa(E_1,E_2) \wedge (b,d) \in \kappa(E_2,E_3)) \implies (a,d) \in \kappa(E_1,E_3)$
4. if $t_2 < t_3$, then
   $(a,b) \in \kappa(E_1,E_2) \iff (a,b) \in \kappa(E_1, E_2 \cup E_3)$
5. if $t_1 < t_2$ or $t_3 < t_2$, then
   $(a,d) \in \kappa(E_1,E_3) \iff ((a,d) \in \kappa(E_1 \cup E_2, E_3) \wedge (a,d) \in \kappa(E_1, E_2 \cup E_3))$
6. if $t_2 < t_1$, then
   $(b,d) \in \kappa(E_2,E_3) \iff (b,d) \in \kappa(E_1 \cup E_2, E_3)$
7. if $t_1 = t_2$, then
   $((O_1 + O_2, t), c) \in \kappa(E_1 \cup E_2, E_3) \iff ((O_1,t),c) \in \kappa(E_1,E_3) \wedge ((O_2,t),c) \in \kappa(E_2,E_3))$
8. if $t_2 = t_3$, then
   $(a,(O_2+O_3,t)) \in \kappa(E_1, E_2 \cup E_3) \iff (a,(O_2,t)) \in \kappa(E_1,E_2) \wedge (a,(O_3,t)) \in \kappa(E_1,E_3))$

We show that

$$\begin{aligned}&((a,b) \in \kappa(E_1,E_2) \wedge (\iota_1(a,b),c) \in \kappa(E_1 \cup E_2, E_3)) \iff \\ &((b,c) \in \kappa(E_2,E_3) \wedge (a, \iota_2(b,c)) \in \kappa(E_1, E_2 \cup E_3))\end{aligned}$$

where, for $o = (O, t)$, $p = (P, l)$, and $k \in \{1, 2\}$, $\iota_k(o, p) = \begin{cases} o & \text{if } t < l \\ p & \text{if } l < t \\ (O +_k P, t) \end{cases}$

Let $o_i = (O_i, t_i) \in TES(E_i)$ with $i \in \{1, 2, 3\}$. We split cases based on time:

– if $t_1 < t_2$, then, using assumption 5 and 1,

$$\begin{aligned} & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o, o_3) \in \kappa(E_1 \cup E_2, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_1, o_3) \in \kappa(E_1 \cup E_2, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_1, o_3) \in \kappa(E_1, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_1, o_3) \in \kappa(E_1, E_3) \wedge (o_2, o_3) \in \kappa(E_2, E_3) \end{aligned}$$

– if $t_1 = t_2$, then, using assumption 7,

$$\begin{aligned} & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o, o_3) \in \kappa(E_1 \cup E_2, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_1, o_3) \in \kappa(E_1, E_3) \wedge (o_2, o_3) \in \kappa(E_2, E_3) \end{aligned}$$

– if $t_2 < t_1$, then, using assumption 6 and 3,

$$\begin{aligned} & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o, o_3) \in \kappa(E_1 \cup E_2, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_2, o_3) \in \kappa(E_1 \cup E_2, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_2, o_3) \in \kappa(E_2, E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_1, o_3) \in \kappa(E_1, E_3) \wedge (o_2, o_3) \in \kappa(E_2, E_3) \end{aligned}$$

Similarly, using the set of assumptions $\{2, 4, 5, 3, 8\}$, the same result can be established about the condition $(o_2, o_3) \in \kappa(E_2, E_3) \wedge (o_1, o') \in \kappa(E_1, E_2 \cup E_3)$, i.e.

$$\begin{aligned} & (o_2, o_3) \in \kappa(E_2, E_3) \wedge (o_1, o') \in \kappa(E_1, E_2 \cup E_3) \\ \iff & (o_1, o_2) \in \kappa(E_1, E_2) \wedge (o_1, o_3) \in \kappa(E_1, E_3) \wedge (o_2, o_3) \in \kappa(E_2, E_3) \end{aligned}$$

We conclude, based on Lemma 5, that the conditions enumerated are sufficient for $\times_{([\kappa],[+])}$ to be associative. □

Proof (Corollary 2). We show the proof for $\times_{(\bowtie_\sqcap,[\cup])}$, and the proof is similar for $\times_{(\not\bowtie_\sqcap,[\cup])}$. We fix the set union $\cup$ as composition function on observables. Let $\sqcap \subseteq \mathcal{P}(\mathbb{E}) \times \mathcal{P}(\mathbb{E})$ be a relation on observables. Let $\bowtie_\sqcap$ be the composability relation defined in Definition 11, and let $\kappa'_{sync,\sqcap}$ be such that, for any $(O_1, t_1) \in \mathcal{P}(E_1) \times \mathbb{R}_+$ and $(O_2, t_2) \in \mathcal{P}(E_2) \times \mathbb{R}_+$:

$$\begin{aligned} ((O_1, t_1), (O_2, t_2)) \in \ & \kappa'_{sync,\sqcap}(E_1, E_2) \iff \\ & (t_1 = t_2 \wedge ((O_1, t_1), (O_2, t_2)) \in \ \kappa_{sync,\sqcap}(E_1, E_2)) \end{aligned}$$

We write $\bowtie'_\sqcap$ for the lifting of $\kappa'_{sync,\sqcap}$.

Commutativity. We first show that if $\sqcap$ is a symmetric relation on observables, then $\kappa_{sync,\sqcap}$ is a symmetric composability relation on observables. Indeed, given the definition of $\kappa_{sync,\sqcap}$ in Definition 11 the synchronous composability

relation is such that for any $(O_1, t_1) \in \mathcal{P}(E_1) \times \mathbb{R}_+$ and $(O_2, t_2) \in \mathcal{P}(E_2) \times \mathbb{R}_+$, $((O_1, t_1), (O_2, t_2)) \in \kappa_{sync,\sqcap}(E_1, E_2)$, if and only if:

$$
\begin{array}{l}
t_1 < t_2 \wedge \neg(\exists O_2' \subseteq E_2.\ (O_1, O_2') \in \sqcap) \vee \\
t_2 < t_1 \wedge \neg(\exists O_1' \subseteq E_1.\ (O_1', O_2) \in \sqcap) \vee \\
t_2 = t_1 \wedge ((O_1, O_2) = (O_1' \cup O_1'', O_2' \cup O_2'') \wedge (O_1', O_2') \in \sqcap \wedge \\
\qquad (\forall O \subseteq E_2.\ (O_1'', O) \notin \sqcap) \wedge (\forall O \subseteq E_1.\ (O, O_2'') \notin \sqcap) \vee (O_1, O_2) = (\emptyset, \emptyset))
\end{array}
$$

Given $\sqcap$ symmetric, we have $((O_1, t_1), (O_2, t_2)) \in \kappa_{sync,\sqcap}(E_1, E_2)$, if and only if:

$$
\begin{array}{l}
t_1 < t_2 \wedge \neg(\exists O_2' \subseteq E_2.\ (O_2', O_1) \in \sqcap) \vee \\
t_2 < t_1 \wedge \neg(\exists O_1' \subseteq E_1.\ (O_2, O_1') \in \sqcap) \vee \\
t_2 = t_1 \wedge ((O_1, O_2) = (O_1' \cup O_1'', O_2' \cup O_2'') \wedge (O_2', O_1') \in \sqcap \wedge \\
\qquad (\forall O \subseteq E_2.\ (O, O_1'') \notin \sqcap) \wedge (\forall O \subseteq E_1.\ (O_2'', O) \notin \sqcap) \vee (O_1, O_2) = (\emptyset, \emptyset))
\end{array}
$$

which is equivalent to $((O_2, t_2), (O_1, t_1)) \in \kappa_{sync,\sqcap}(E_2, E_1)$. Thus, if $\sqcap$ is symmetric, then $\kappa_{sync,\sqcap}$ is symmetric.

Associativity. Let $\sqcap$ be the smallest relation such that, for all $O_1, O_2, O_3 \in \mathbb{E}$, $(O_1 \cup O_2, O_3) \in \sqcap \iff ((O_1, O_3) \in \sqcap \wedge (O_2, O_3) \in \sqcap)$ and $(O_1, O_2 \cup O_3) \in \sqcap \iff ((O_1, O_3) \in \sqcap \wedge (O_2, O_3) \in \sqcap)$. We show that the constraint for $\sqcap$ are sufficient to make $\kappa'_{sync,\sqcap}$ satisfying the assumptions in Lemma 7. We first observe that $\kappa'_{sync,\sqcap}$ enforces observations to have the same time. Therefore, it is sufficient to prove that $\kappa'_{sync,\sqcap}$ satisfies assumption 7 and 8 of Lemma 7 to conclude for the associativity of $\times_{(\bowtie'_\sqcap, [\cup])}$.

We show that assumption 7 is satisfied. In the case where $O_1 = O_2 = O_3 = \emptyset$, the assumption 7 is trivially satisfied. For any non empty observations $a = (O_1, t_1) \in \mathcal{P}(E_1) \times \mathbb{R}_+$, $b = (O_2, t_2) \in \mathcal{P}(E_2) \times \mathbb{R}_+$, and $c = (O_3, t_3) \in \mathcal{P}(E_3) \times \mathbb{R}_+$, if $((O_1 \cup O_2, t_1), (O_3, t_3)) \in \kappa'_{(sync,\sqcap)}(E_1 \cup E_2, E_3)$, then we know that there exists O and O' such that $O_1 \cup O_2 = O \cup O'$ and O_3' and O_3'' such that $O_3 = O_3' \cup O_3''$ with:

- $(O, O_3') \in \sqcap$
- $\forall P \subseteq E_3.\ (O', P) \notin \sqcap$, and
- $\forall Q \subseteq E_1 \cup E_2.\ (Q, O_3'') \notin \sqcap$

using assumption on $\sqcap$, we have:

- $(O, O_3') \in \sqcap$ implies that $(O \cap O_2, O_3') \in \sqcap$,
- $\forall P \subseteq E_3.\ (O', P) \notin \sqcap$ implies that $\forall P \subseteq E_3.\ (O' \cap O_2, P) \notin \sqcap$, and
- $\forall Q \subseteq E_1 \cup E_2.\ (Q, O_3'') \notin \sqcap$ implies that $\forall Q \subseteq E_2.\ (Q, O_3'') \notin \sqcap$

as a consequence, $((O_2, t_2), (O_3, t_3)) \in \kappa'_{(sync,\sqcap)}(E_2, E_3)$.

The proof is similar for assumption 8, and we can conclude that $\kappa'_{(sync,\sqcap)}$ satisfies the sufficient condition for the product $\times_{(\bowtie'_\sqcap, [\cup])}$ to be associative.

Idempotency. Let $\sqcap$ be the smallest relation such that for all non empty observables $O \subseteq \mathbb{E}$, $(O, O) \in \sqcap$. We show that $(\sigma, \tau) \in \kappa_{sync,\sqcap}(E, E) \implies \sigma = \tau$.

Indeed, $((O_1, t_1), (O_2, t_2)) \in \kappa_{sync,\sqcap}(E, E)$ if and only if

$$
\begin{aligned}
&t_1 < t_2 \wedge \forall O_2' \subseteq E.\ (O_1, O_2') \notin \sqcap \vee \\
&t_2 < t_1 \wedge \forall O_1' \subseteq E.\ (O_1', O_2) \notin \sqcap \vee \\
&t_2 = t_1 \wedge ((O_1, O_2) = (O_1' \cup O_1'', O_2' \cup O_2'') \wedge (O_1', O_2') \in \sqcap \wedge \\
&\qquad ((\forall O \subseteq E.\ (O_1'', O) \in \sqcap) \wedge (\forall O \subseteq E.\ (O, O_2'') \in \sqcap)) \vee (O_1, O_2) = (\emptyset, \emptyset))
\end{aligned}
$$

First, observe that the constraints imposed on $\sqcap$ implies that for any non empty observable $O \subseteq E$ there exists O such that $(O, O) \in \sqcap$. Hence, the two first disjuncts are necessarily false, and $((O_1, t_1), (O_2, t_2)) \in \kappa_{sync,\sqcap}(E, E)$ implies that $t_1 = t_2$. Then, the only case where the third disjunct is true is when $O_1'' = O_2'' = \emptyset$, in which case $O_1 = O_2$. □

A Multi-Agent Model for Polarization under Confirmation Bias in Social Networks ⋆

Mário S. Alvim[1], Bernardo Amorim[1], Sophia Knight[2], Santiago Quintero[3], and Frank Valencia[4,5]

[1] Department of Computer Science, UFMG, Brazil
[2] Department of Computer Science, University of Minnesota Duluth, USA
[3] LIX, École Polytechnique de Paris, France
[4] CNRS-LIX, École Polytechnique de Paris, France
[5] Pontificia Universidad Javeriana Cali, Colombia

Abstract. We describe a model for polarization in multi-agent systems based on Esteban and Ray's standard measure of polarization from economics. Agents update their beliefs/opinions based on an underlying influence graph, as in the standard DeGroot model for social learning, but under *confirmation bias*: discounting the opinions of agents with dissimilar views. We investigate the conditions where polarization does or does not converge to zero. This work is presented in greater detail at https://arxiv.org/abs/2104.11538. It has been accepted at FORTE 2021.

1 Introduction

Social networks facilitate the exchange of opinions by providing users with information from influencers, friends, or other users with similar or sometimes opposing views [2]. This may allow a healthy exposure to diverse perspectives. On the other hand but social networks may lead users to the problems of cognitive biases, misinformation and radicalization of opinions. It has become evident that a better understanding of social networks is crucial, and there is a growing interest in the development of models for the analysis of polarization and social influence in networks [3–5, 7, 9–12, 14, 15, 17–19].

The Model. We presented our multi-agent model for polarization in [1], inspired by linear-time models of concurrency where the state of the system evolves in discrete time units (in particular [13, 16]). At each time unit, the agents *update* their beliefs about the proposition of interest taking into account the beliefs of their neighbors in an underlying weighted *influence graph*. The belief update gives more value to the opinion of agents with higher influence (*authority bias*) and to the opinion of agents with similar views (*confirmation bias*). Furthermore, the model is equipped with a *polarization measure* based on the seminal work in economics by Esteban and Ray [8]. The polarization is measured at each time unit and it is 0 if all agents' beliefs fall within an interval of agreement about the proposition.

In the current paper we study the network conditions, subject to confirmation bias, under which polarization vanishes or remains forever, and prove these results. The main goal of this paper is identifying how networks and beliefs are structured, for agents subject to confirmation bias, when polarization *does not* disappear. Our results provide insight into the phenomenon of polarization, and are a step toward the design of robust computational models and simulation software for human cognitive and social processes.

2 The Model

Here we refine the polarization model introduced in [1], composed of static and dynamic elements. We presuppose basic knowledge of calculus and graph theory [6, 20].

⋆ Mário S. Alvim and Bernardo Amorim were partially supported by CNPq, CAPES and FAPEMIG. Santiago Quintero and Frank Valencia were partially supported by the ECOS-NORD project FACTS (C19M03).

Static Elements of the Model *Static elements* of the model represent a snapshot of a social network at a given point in time. They include a set $\mathcal{A}$ of *agents*, a *proposition* p of interest, about which agents can hold beliefs, and a *belief configuration* $B{:}\mathcal{A}\to[0,1]$ s.t. B_i is the current confidence of agent $i\in\mathcal{A}$ in proposition p. 0 and 1 represent a firm belief in, respectively, the falsehood or truth of p.

A *polarization measure* $\rho{:}[0,1]^{\mathcal{A}}\to\mathbb{R}$ maps belief configurations to real numbers.

We use Esteban-Ray Polarization [8], denoted $\rho_{ER}(\pi, y)$ where π and y represent the current belief state. This value indicates how polarized the belief configuration is; the higher the value of $\rho_{ER}(\pi, y)$, the more polarized distribution (π, y) is. Details may be found in the full version of the paper.

When there is consensus about the proposition p of interest, i.e., when all agents in belief configuration hold the same belief value, we have $\rho(B)=0$.

Dynamic Elements of the Model *Dynamic elements* formalize the evolution of agents' beliefs as they interact over time and are exposed to different opinions. They include:

- A *weighted directed graph* $\mathcal{I}{:}\mathcal{A}\times\mathcal{A}\to[0,1]$. The value $\mathcal{I}_{i,j}$, represents the *direct influence* that agent i has on agent j, orthe *trust* or *confidence* that j has in i.
- An *update function* $\mu{:}(B^t,\mathcal{I})\mapsto B^{t+1}$ mapping belief configuration B^t at time t and influence graph $\mathcal{I}$ to new belief configuration B^{t+1} at time $t{+}1$. This function models the evolution of agents' beliefs over time.

Now we define our specific update function, based on the idea that agents use *confirmation bias*: each agent is most influenced by other agents whose beliefs are close to the beliefs that agent already holds. In earlier work we studied other methods of belief update.

Definition 1 (Confirmation-bias). *Let B^t be a belief configuration at time $t\in\mathcal{T}$, and $\mathcal{I}$ be an influence graph. The* confirmation-bias update-function *is the map $\mu^{CB}{:}(B^t,\mathcal{I})\mapsto B^{t+1}$ with B^{t+1} given by $B_i^{t+1} = B_i^t + 1/|\mathcal{A}_i| \sum_{j\in\mathcal{A}_i} \beta_{i,j}^t \mathcal{I}_{j,i} (B_j^t - B_i^t)$, for every agent $i\in\mathcal{A}$, where $\mathcal{A}_i = \{j\in\mathcal{A} \mid \mathcal{I}_{j,i}>0\}$ is the set of* neighbors *of i and $\beta_{i,j}^t = 1-|B_j^t - B_i^t|$ is the* confirmation-bias factor *of i w.r.t. j given their beliefs at time t.*

The confirmation-bias factor is $\beta_{i,j}^t \in [0,1]$. The lower its value, the more agent i discounts the opinion provided by agent j when incorporating it. It is maximum when agents' beliefs are identical, and minimum when they are extremely different.

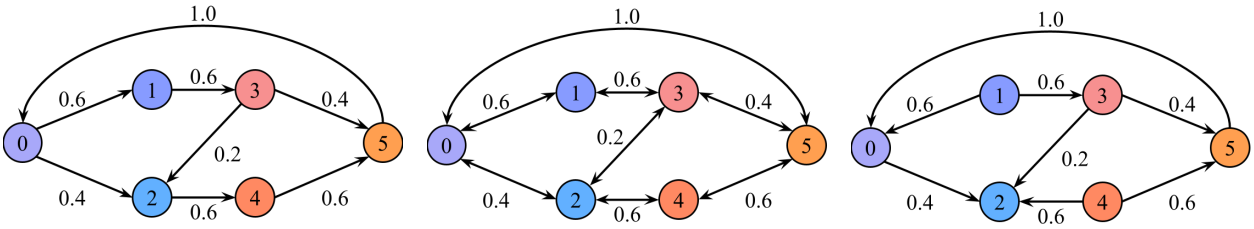
2.1 Running Example and Simulations

We now present a running example and several simulations that motivate our theoretical results.

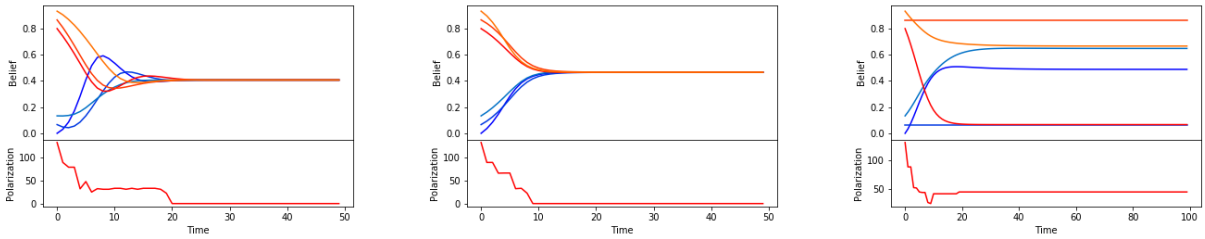
Example 1 (Vaccine Polarization). Consider the sentence "vaccines are safe" as the proposition of interest, p. Assume a set $\mathcal{A}$ of 6 agents that is initially *extremely polarized* about p: agents 0 and 5 are absolutely confident, respectively, in the falsehood or truth of p, whereas the others are equally split into strongly in favour and strongly against p.

Consider the situation in Fig. 1a. Nodes 0, 1 and 2 represent anti-vaxxers, whereas the rest are pro-vax. In particular, although initially in total disagreement about p, Agent 5 carries a lot of weight with Agent 0. In contrast, Agent 0's opinion is very close to that of 1 and 2, even if they do not have any direct influence over him. Hence the evolution of Agent 0's beliefs will be mostly shaped by that of Agent 5. As can be observed in the evolution of agents' opinions in Fig. 1d, Agent 0 moves from being strongly against to being fairly in favour of p around time step 8. Moreover, polarization eventually vanishes around time 20, as agents reach the consensus of being fairly against p.

Now consider the influence graph in Fig. 1b, which is similar to Fig. 1a, but with reciprocal influences (i.e., the influence of i over j is the same as the influence of j over i). Now Agents 1 and 2 do have direct influences over Agent 0, so the evolution of Agent 0's belief will be partly shaped by initially opposed agents: Agent 5 and the anti-vaxxers. But since Agent 0's opinion is very close to that of Agents 1 and 2, the confirmation-bias factor will help keeping Agent 0's opinion close to their opinion against p. In particular, in contrast to the situation in Fig. 1d, Agent 0 never becomes in favour of p. The evolution of the agents' opinions and their polarization is shown in Fig. 1e. Notice that polarization vanishes around time 8 as the agents reach consensus but this time they are more positive about (less against) p than in the first situation.



(a) Influence graph $\mathcal{I}$ for Ex. 1. (b) Adding inverse influences to Fig. 1a. (c) Inversion of $\mathcal{I}_{0,1}$ and $\mathcal{I}_{2,4}$ in Fig. 1a.



(d) Beliefs and pol. for Fig. 1a. (e) Beliefs and pol. for Fig. 1b. (f) Belief and pol. for Fig. 1c.

Fig. 1: Influence graphs and evolution of beliefs and polarization for Ex. 1.

Finally, consider the situation in Fig. 1c obtained from Fig. 1a by inverting the influences of Agent 0 over Agent 1 and Agent 2 over Agent 4. Notice that Agents 1 and 4 are no longer influenced by anyone though they influence others. Thus, as shown in Fig.1f, their beliefs do not change over time, which means that the group does not reach consensus and polarization never disappears though it is considerably reduced. □

The above example illustrates complex non-monotonic, overlapping, convergent, and non-convergent evolution of agent beliefs and polarization even in a small case with n=6 agents. In the full paper, we present simulations for several influence graph topologies with n=1 000 agents, which illustrate more of this complex behavior emerging from confirmation-bias interaction among agents. Our theoretical results in the next sections bring insight into the evolution of beliefs and polarization depending on graph topologies.

3 Belief and Polarization Convergence

Polarization diminishes as agents approximate a *consensus*, i.e., as they (asymptotically) agree upon a common belief value for the proposition of interest[6]. Here and in Section 4 we consider meaningful families of influence graphs that guarantee consensus *under confirmation bias*. We relate influence with the notion of *flow* in flow networks, and use it to identify necessary conditions for polarization not converging to zero.

3.1 Convergence under Confirmation Bias in Strongly Connected Influence

We now introduce the family of *strongly-connected* influence graphs, which includes cliques, that describes scenarios where each agent has an influence, possibly indirect, over each other agent.

Definition 2 (Influence Paths). *Let $C \in (0, 1]$. We say that i has a* direct influence *C over j, written $i \xrightarrow{C} j$, if $\mathcal{I}_{i,j} = C$.*

An influence path *is a finite sequence of* distinct *agents from $\mathcal{A}$ where each agent in the sequence has a direct influence over the next one. Let p be an influence path $i_0 i_1 \dots i_n$. The* size *of p is $|p|$=n. We also use $i_0 \xrightarrow{C_1} i_1 \xrightarrow{C_2} \dots \xrightarrow{C_n} i_n$*

[6] In the extended version of this paper, we discuss one unusual situation where this does not occur, but in all normal situations polarization diminishes as agents approach consensus.

to denote p with the direct influences along this path. We write $i_0 \overset{C}{\rightsquigarrow}_p i_n$ to indicate that the product influence *of i_0 over i_n along p is $C{=}C_1{\times}\ldots{\times}C_n$.*

We often omit influence or path indices from the above arrow notations when they are unimportant or clear from the context. We say that i has an influence *over j if $i{\rightsquigarrow}j$.*

The next definition is akin to the graph-theoretical notion of strong connectivity.

Definition 3 (Strongly Connected Influence). *We say that an influence graph $\mathcal{I}$ is* strongly connected *if for all i, $j{\in}\mathcal{A}$ such that $i{\neq}j$, $i{\rightsquigarrow}j$.*

Remark 1. For technical reasons we assume that, *initially*, there are no two agents $i, j{\in}\mathcal{A}$ such that $B^0_i{=}0$ and $B^0_j{=}1$. This implies that for every $i, j{\in}\mathcal{A}$: $\beta^0_{i,j}{>}0$ where $\beta^0_{i,j}$ is the confirmation bias of i towards j at time 0 (See Def. 1). Nevertheless, at the end of this section we will address the cases in which this condition does not hold.

Definition 4 (Radical Beliefs). *An agent $i{\in}\mathcal{A}$ is called* radical *if $B_i{=}0$ or $B_i{=}1$. A belief configuration B is* radical *if every $i{\in}\mathcal{A}$ is radical.*

Theorem 1 (Confirmation-Bias Belief Convergence). *In a strongly connected influence graph and under the confirmation-bias update-function, if B^0 is not radical then for all $i, j{\in}\mathcal{A}$, $\lim_{t\to\infty} B^t_i{=}\lim_{t\to\infty} B^t_j$. Otherwise for every $i{\in}\mathcal{A}$, $B^t_i{=}B^{t+1}_i{\in}\{0,1\}$.*

We conclude this section by emphasizing that belief convergence is not guaranteed in non strongly-connected graphs. Fig. 1c from the vaccine example shows such a graph where neither belief convergence nor zero-polarization is obtained.

4 Conditions for Polarization

We identify necessary conditions under confirmation bias for polarization to persist, one of our main contributions.

Balanced Influence: Circulations The following is inspired by the *circulation problem* for directed graphs [6].

Definition 5 (Balanced Influence). *We say that $\mathcal{I}$ is* balanced *(or a* circulation*) if every $i \in \mathcal{A}$ satisfies the constraint $\sum_{j\in\mathcal{A}} \mathcal{I}_{i,j}{=}\sum_{j\in\mathcal{A}} \mathcal{I}_{j,i}$.*

Cliques and circular graphs, where all (non-self) influence values are equal, are balanced. The graph of our vaccine example (Fig. 1) is a circulation that it is neither a clique nor a circular graph.

Next we use a fundamental property from flow networks describing flow conservation for graph cuts [6]. Interpreted in our case it says that any group of agents $A{\subseteq}\mathcal{A}$ influences other groups as much as they influence A.

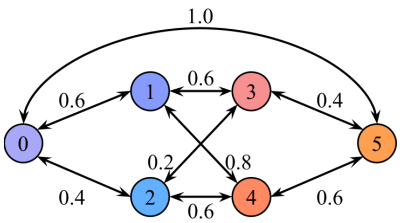
Proposition 1 (Group Influence Conservation). *Let $\mathcal{I}$ be balanced and $\{A, B\}$ be a partition of $\mathcal{A}$. Then $\sum_{i\in A}\sum_{j\in B}\mathcal{I}_{i,j} = \sum_{i\in A}\sum_{j\in B}\mathcal{I}_{j,i}$.*

We now define *weakly connected influence*. Recall that an undirected graph is *connected* if there is path between each pair of nodes.

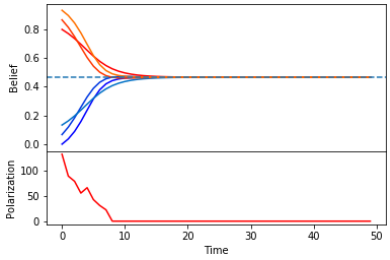
Definition 6 (Weakly Connected Influence). *Given an influence graph $\mathcal{I}$, define the undirected graph $G_{\mathcal{I}}{=}(\mathcal{A}, E)$ where $\{i, j\}{\in}E$ if and only if $\mathcal{I}_{i,j}{>}0$ or $\mathcal{I}_{j,i}{>}0$. An influence graph $\mathcal{I}$ is called* weakly connected *if the undirected graph $G_{\mathcal{I}}$ is connected.*

Weakly connected influence relaxes its strongly connected counterpart. However, every balanced, weakly connected influence is strongly connected as implied by the next lemma. Intuitively, circulation flows never leaves strongly connected components.

Lemma 1. *If $\mathcal{I}$ is balanced and $\mathcal{I}_{i,j}{>}0$ then $j{\rightsquigarrow}i$.*



(a) Regular and reciprocal influence.



(b) Beliefs and pol. for Fig. 2a.

Fig. 2: Influence and evolution of beliefs and polar.

Conditions for Polarization We now have all elements to identify conditions for permanent polarization.

Theorem 2 (Conditions for Polarization). *Suppose that* $\lim_{t\to\infty} \rho(B^t)\neq 0$. *Then either: (1)* $\mathcal{I}$ *is not balanced; (2)* $\mathcal{I}$ *is not weakly connected; or (3)* B^0 *is radical; or (4) for some borderline value* v, $\lim_{t\to\infty} B_i^t = v$ *for each* $i\in\mathcal{A}$[7] .

Hence, at least one of the four conditions is necessary for the persistence of polarization. If (1) then there must be at least one agent that influences more than he is influenced (or vice versa). This is illustrated in Fig. 1c from the vaccine example, where Agent 2 is such an agent. If (2) then there must be isolated subgroups of agents; e.g., two isolated strongly-connected components the members of the same component will achieve consensus but the consensus values of the two components may be very different. Condition (3) can be ruled out if there is an agent that is not radical, like in all of our examples and simulations. (4) depends on technical details of the polarization measure and rarely arises.

Reciprocal and Regular Circulations The notion of circulation allowed us to identify potential causes of polarization. In this section we will also use it to identify meaningful topologies whose symmetry can help us predict the exact belief value of convergence.

A *reciprocal* influence graph is a circulation where the influence of i over j is the same as that of j over i, i.e, $\mathcal{I}_{i,j}=\mathcal{I}_{j,i}$. Also a graph is (*in-degree*) *regular* if the in-degree of each nodes is the same; i.e., for all $i, j\in\mathcal{A}$, $|\mathcal{A}_i|=|\mathcal{A}_j|$.

As examples of regular and reciprocal graphs, consider a graph $\mathcal{I}$ where all (non-self) influence values are equal. If $\mathcal{I}$ is *circular* then it is a regular circulation, and if $\mathcal{I}$ is a *clique* then it is a reciprocal regular circulation. Also we can modify slightly our vaccine example to obtain a regular reciprocal circulation as shown in Fig. 2.

Theorem 3 (Consensus Value). *Suppose that* $\mathcal{I}$ *is regular and weakly connected. If* $\mathcal{I}$ *is reciprocal and the belief update is confirmation-bias, or if the influence graph* $\mathcal{I}$ *is a circulation and the belief update is classical, then* $\lim_{t\to\infty} B_i^t = {}^1/_{|\mathcal{A}|} \sum_{j\in\mathcal{A}} B_j^0$ *for every* $i\in\mathcal{A}$.

5 Conclusions and Future Work

We proposed a model for polarization and belief evolution for multi-agent systems under confirmation-bias. We showed that whenever all agents can directly or indirectly influence each other, their beliefs always converge, and so does polarization as long as the convergence value is not a borderline point. We also identified necessary conditions for polarization not to disappear, and the convergence value for some important network topologies. As future work we intend to extend our model to model evolution of beliefs and measure polarization in situations in which agents hold opinions about multiple propositions of interest.

[7] This case refers to the unusual and technical conditions under which polarization can persist, even when agents' beliefs converge.

Bibliography

[1] Alvim, M.S., Knight, S., Valencia, F.: Toward a formal model for group polarization in social networks. In: The Art of Modelling Computational Systems. Lecture Notes in Computer Science, vol. 11760, pp. 419–441. Springer (2019)

[2] Bozdag, E.: Bias in algorithmic filtering and personalization. Ethics and Information Technology (09 2013)

[3] Calais Guerra, P., Meira Jr, W., Cardie, C., Kleinberg, R.: A measure of polarization on social media networks based on community boundaries. Proceedings of the 7th International Conference on Weblogs and Social Media, ICWSM 2013 pp. 215–224 (01 2013)

[4] Christoff, Z., et al.: Dynamic logics of networks: information flow and the spread of opinion. Ph.D. thesis, PhD Thesis, Institute for Logic, Language and Computation, University of Amsterdam (2016)

[5] DeGroot, M.H.: Reaching a consensus. Journal of the American Statistical Association **69**(345), 118–121 (1974)

[6] Diestel, R.: Graph Theory. Springer-Verlag, fifth ed edn. (2015)

[7] Elder, A.: The interpersonal is political: unfriending to promote civic discourse on social media. Ethics and Information Technology pp. 1–10 (2019)

[8] Esteban, J.M., Ray, D.: On the measurement of polarization. Econometrica **62**(4), 819–851 (1994)

[9] Gargiulo, F., Gandica, Y.: The role of homophily in the emergence of opinion controversies. arXiv preprint arXiv:1612.05483 (2016)

[10] Golub, B., Jackson, M.O.: Naive learning in social networks and the wisdom of crowds. American Economic Journal: Microeconomics **2**(1), 112–49 (2010)

[11] Hunter, A.: Reasoning about trust and belief change on a social network: A formal approach. In: International Conference on Information Security Practice and Experience. pp. 783–801. Springer (2017)

[12] Li, L., Scaglione, A., Swami, A., Zhao, Q.: Consensus, polarization and clustering of opinions in social networks. IEEE Journal on Selected Areas in Communications **31**(6), 1072–1083 (2013)

[13] Nielsen, M., Palamidessi, C., Valencia, F.D.: Temporal concurrent constraint programming: Denotation, logic and applications. Nord. J. Comput. **9**(1), 145–188 (2002)

[14] Pedersen, M.Y.: Polarization and echo chambers: A logical analysis of balance and triadic closure in social networks

[15] Proskurnikov, A.V., Matveev, A.S., Cao, M.: Opinion dynamics in social networks with hostile camps: Consensus vs. polarization. IEEE Transactions on Automatic Control **61**(6), 1524–1536 (June 2016)

[16] Saraswat, V.A., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: LICS. pp. 71–80. IEEE Computer Society (1994)

[17] Seligman, J., Liu, F., Girard, P.: Logic in the community. In: Indian Conference on Logic and Its Applications. pp. 178–188. Springer (2011)

[18] Seligman, J., Liu, F., Girard, P.: Facebook and the epistemic logic of friendship. CoRR **abs/1310.6440** (2013)

[19] Sîrbu, A., Pedreschi, D., Giannotti, F., Kertész, J.: Algorithmic bias amplifies opinion polarization: A bounded confidence model. arXiv preprint arXiv:1803.02111 (2018)

[20] Sohrab, H.H.: Basic Real Analysis. Birkhauser Basel, 2nd ed edn. (2014)

Expanding Social Polarization Models by Incorporating Belief Credibility

Alex Chambers[1] and Sophia Knight[2]

[1,2]University of Minnesota Duluth, USA

May 2021

Abstract

We present work in progress on modelling social networks which include agents with quantitative beliefs about multiple issues, including interconnections between an agent's own beliefs, and the effects of these interconnections on the evolution of the system as a whole. This work refines the formal models for social networks presented in [1, 2].

1 Introduction

As social networks become ubiquitous in our lives and have unpredicted and unprecedented effects on communication and public opinion, it is crucial to develop accurate formal models to better understand them. In [1, 2], a formal model of social networks was developed, including a quantitative measure of agents' beliefs about a single issue, their change over time, and the overall polarization of the social network at each time step. Here, we present work in progress aimed at refining this model by allowing agents to have beliefs about multiple issues, and examining the influence between an agent's belief about one issue and a related issue.

2 Contribution

Although the previous model provided a way to simulate the state of single belief, it cannot capture many of the nuances present in those interactions. This was cause to develop a new flexible update function that was able to fill in the gaps and provide a more accurate model.

This model provides a major improvement over the previous one in terms of functionality. The first component is the transition to a multiple belief update system to represent a wide spectrum of interconnected beliefs. Similarly to before, each belief will represent an agent's view or stance on the corresponding

topic. However, in real interactions, one's beliefs are rarely independent, and instead often depend on each other to form an overall narrative or stance.

Implementing this factor can be done through aggregation of related beliefs. This is done through the use of a belief influence table. This table causes each belief to affect the expression of others and establish a ***credibility*** measure for each of an agent's beliefs. This allows for situations where multiple similar beliefs may support each other, therefore increasing the expression of that belief; as well as situations where contradicting beliefs limit an agent's influence. This credibility measurement replaces the influence multiplier from the previous model. We also add a fortitude multiplier, used to resist incoming credibility, and therefore resist being influenced.

Credibility of a belief k, denoted as $C[k]$ is established as follows:

$$C[k] = \sum_{i=1}^{n} B[i] * T[k, i]$$

where,

n is the number of beliefs in the system,

B is the list of an agent's beliefs,

C is the list of an agent's credibilites, and

T is the influence table, with $T[k, i]$ representing the percentage of belief i that applies to belief k.

The existing update method does not support this credibility measurement, so we need to replace it with something that does. Due to its nature, this new function remaps beliefs from [0,1] to [-1,1] and has multiple useful properties:

1. An agent with a credibility of 0 exerts no influence.
2. An agent with a neutral belief of 0 exerts no influence
3. A polarized agent is less receptive to opposing influence
4. A polarized agent more easily becomes increasingly polarized

The function to calculate h_k, the change in an agent a_1's belief k when influenced by agent a_i is defined as follows:

$$h_k = \frac{x_1 + \sqrt[3]{cx_2 - x_1^3}}{s}$$

where,

$x_1 = B_{a_1}[k]$,

$x_2 = B_{a_i}[k]$,

$f = a_i$'s fortitude multiplier, or resistance to influence,

$c = C_{a_2}[k] * f$, and

s is an arbitrary scalar.

Agent a_1's new belief $N[k]$ from this step is then incremented by $h_1 + x_1$. This update function is run for every influencing agent in the population, then the agent's beliefs x_1 are set to the average of the new beliefs, $\frac{N}{n}$.

With these new functions, a simulation can be run to model the changes in beliefs across a system of agents. Each update step consists of the following:

1. Apply the update function on each agent.
2. Average the agent's new beliefs based on the number of influencing agents.
3. Overwrite each agent's old beliefs with their new ones.
4. Calculate the credibility of each agent's beliefs.

After each update step, a snapshot of the system's polarization can be stored.

3 Example

Let's consider one influence system, containing just a few beliefs.

1. A **pivot** belief that will be the focus of the simulation,
2. A **tethered** belief that is mostly or entirely affected by the pivot belief,
3. A strong **external** belief that is used for constant, immutable influence on the pivot, and
4. A **side** belief that is able to affect the pivot better than the "external belief.

Suppose we have a system with a balanced population composed of agents holding only mildly strong, opposing **pivot** beliefs who can all slightly influence each other. Over time, their polarization will remain constant, as each agent has just as much reinforcement as it does opposition.

Now, let's add a single incredibly influential agent to the population. This could be someone such as a family member or politician. In general, this will represent someone with a strong influence that is difficult or impossible to change. This represents our **external** belief and can be conceptualized with something such as agreement with a news station.

Without resistance, the influential agent will align the other agents' **external** beliefs to match their own, in turn causing the **pivot** and the **tether** beliefs to follow. This results in a polarized system.

However, if we add just a single opposing agent, possessing a **side** belief, one that the influential agent cannot affect, the situation changes. At first, it begins the same as before with agents converging to the influential agent's belief. However, as the **side** belief spreads, it eventually overpowers the **external** belief, causing the pivot to flip and bringing the agents to the opposite belief.

This technique is one that is often used in discussion of difficult or controversial topics. Instead of directly discussing a topic that will cause both parties to become upset or reluctant to listen, the conversation can more safely be directed toward a common ground of a related topic. As a result, heated and divisive subjects can still be discussed without as easily resorting to bias or being restricted behind polarized views.

To facilitate expansion and simulation of the model, the previous simulation was rewritten using C# and then expanded upon to incorporate the new systems. The current program contains high level structures for creating, manipulating, and simulating large populations of agents with various influence maps and beliefs. This tool was used to run a simulation of the system described above and track how it behaves over time.

The initial population is as follows:

"politician" agent with beliefs [1, 1, 1, 0], fortitude 1

"citizen" agents with beliefs [1, 1, 0, 0], fortitude 0

"citizen" agents with beliefs [-1, -1, 0, 0], fortitude 0

"citizen" agent with beliefs [-1, -1, 0, -0.3], fortitude 0

The influence table:

-	b_{1out}	b_{2out}	b_{3out}	b_{4out}
b_{1in}	0	0	0.2	0.8
b_{2in}	0.5	0	0	0
b_{3in}	0	0	0.2	0
b_{4in}	0	0	0	0.1

Where belief 1 is the **pivot**, belief 2 is the **tether**, belief 3 is the **external**, and belief 4 is the **side**. As you can see, belief 3, the **external** belief, and belief 4, the **side** belief both affect the **pivot** belief, 1, though belief 4 has a much greater effect.

For the first 50 steps, beliefs 1 and 2 remain highly polarized, as there are no significant influences acting to change them. During this time however, belief 3 gradually becomes less polarized as agents are influenced by the politician's **external** belief. By step 130, all agents are in consensus with the politician

on belief 3, causing the **pivot** to follow suit. During this time, belief 4 is very slowly spreading throughout the system.

After some time, around step 480, the agents are nearly in consensus with the politician on the **pivot** belief, and the **tether** belief, belief 2, is following closely behind. However, it is at this point that belief 4 has built up enough to suddenly flip the pivot back around. The agents reverse their beliefs and the pivots are now in line with that of the original agent holding the **side** belief. The **tether** belief then follows.By step 510, the **pivot** belief has stabilized, and by step 570, the **tether** has done the same. After this point, the system remains unchanged.

4 Related Work

There is a great deal of work on social networks, in various fields. Here we briefly discuss the most closely related work involving formal models of social networks.

DeGroot Models The closest related work is DeGroot models [3]. These are the standard linear models for social learning whose analysis can be carried out by linear techniques from Markov chains, and are a similar approach to that of [2, 1]. In the current work we extend this class of models by analyzing the effects of interacting beliefs about multiple issues for each agent.

Other Formal Models Sîrbu et al. [5] use a formal model of social networks that updates probabilistically to investigate the effects of algorithmic bias on polarization by counting the number of opinion clusters, interpreting a single opinion cluster as consensus. Leskovec et al. [4] simulate social networks and observe group formation over time.

5 Conclusion and Future Work

Formal models of social networks are important, in order to better understand and predict the effects of social networks on widespread opinions and beliefs. Here, we have presented work in progress on formal models where agents may have multiple, interdependent, quantitative beliefs about a variety of topics. This is valuable for making our formal models more realistic and representative of reality.

Future Work Our immediate plans are to further refine our parameters and run more simulations, in order to improve our model and its ability to model realistic situations. Then we will be able to understand and prove general properties in our models, and compare them to real-world data about social networks.

References

[1] Alvim, M. S., Amorim, B., Knight, S., Quintero, S., and Valencia, F. A multi-agent model for polarization under confirmation bias in social networks. *CoRR abs/2104.11538* (2021).

[2] Alvim, M. S., Knight, S., and Valencia, F. Toward a formal model for group polarization in social networks. In *The Art of Modelling Computational Systems* (2019), vol. 11760 of *LNCS*, Springer, pp. 419–441.

[3] DeGroot, M. H. Reaching a consensus. *Journal of the American Statistical Association 69*, 345 (1974), 118–121.

[4] Gargiulo, F., and Gandica, Y. The role of homophily in the emergence of opinion controversies. *arXiv preprint arXiv:1612.05483* (2016).

[5] Sîrbu, A., Pedreschi, D., Giannotti, F., and Kertész, J. Algorithmic bias amplifies opinion polarization: A bounded confidence model. *arXiv preprint arXiv:1803.02111* (2018).