Science of Computer Programming 115–116 (2016) 47–78

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A procedure for splitting data-aware processes and its application to coordination ☆

S.-S.T.Q. Jongmans [a,*], D. Clarke [b], J. Proença [b]

[a] Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, Netherlands
[b] Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium

HIGHLIGHTS

- We present a procedure for splitting algebraic processes with multiactions and data.
- We prove its correctness (strong bisimilarity between original and split processes).
- We apply it to the process algebraic semantics of the coordination language Reo.
- This application justifies an optimization technique for Reo implementations.

ARTICLE INFO

Article history:
Received 3 February 2013
Received in revised form 31 August 2013
Accepted 13 February 2014
Available online 28 February 2014

Keywords:
Process algebra
Coordination
mCRL2
Reo

ABSTRACT

We present a procedure for splitting processes in a process algebra with multiactions and data (the untimed subset of the specification language mCRL2). This splitting procedure cuts a process into two processes along a set of actions A: roughly, one of these processes contains no actions from A, while the other process contains only actions from A. We state and prove a theorem asserting that the parallel composition of these two processes is provably equal from a set of axioms (sound and complete with respect to strong bisimilarity) to the original process under some appropriate notion of synchronization.
We apply our splitting procedure to the process algebraic semantics of the coordination language Reo: using this procedure and its related theorem, we formally establish the soundness of splitting Reo connectors along the boundaries of their (a)synchronous regions in implementations of Reo. Such splitting can significantly improve the performance of connectors as shown elsewhere.

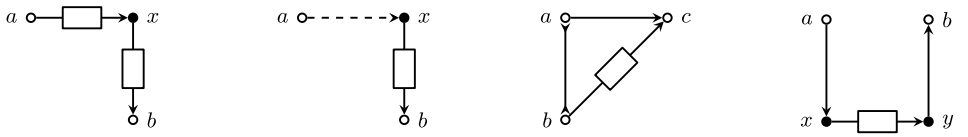
1. Motivation

Context Over the past decades, coordination languages have emerged for the specification and implementation of interaction protocols among entities running concurrently (components, services, threads, etc.). This class of languages includes Reo [2,3], a graphical language for compositional construction of *connectors*: communication media through which entities can interact with each other. Fig. 1 shows some example Reo connectors in their usual graphical syntax. Intuitively, connectors consist of one or more *channels* (i.e., the edges of a connector graph), through which data items flow, and a number of *nodes* (i.e., the vertices of a connector graph), on which channel *ends* (i.e., the endpoints of edges) meet. Through channel

☆ This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software Using Formal Models (http://www.hats-project.eu/).

* Corresponding author.
E-mail addresses: jongmans@cwi.nl (S.-S.T.Q. Jongmans), dave.clarke@cs.kuleuven.be (D. Clarke), jose.proenca@cs.kuleuven.be (J. Proença).

http://dx.doi.org/10.1016/j.scico.2014.02.017
0167-6423/



(a) FIFO2 (b) LossyFIFO (c) Alternator (d) SyncFIFOSync

Fig. 1. Example connectors.

composition—the act of gluing channels together on nodes—engineers can construct complex connectors. Channels often used include the reliable synchronous channel, called sync, and the reliable asynchronous channel fifo$_n$, which has a buffer of capacity n. Importantly, while nodes have a fixed semantics, Reo features an open-ended set of channels. This allows engineers to define their own channels with custom semantics.

To use connectors in real applications, one must derive executable code from graphical specifications of connectors (e.g., those in Fig. 1). Roughly two implementation approaches currently exist. In the *distributed approach* [15,45,43,44], one implements the behavior of each of the k constituents of a connector and runs these k implementations concurrently as a distributed system; in the *centralized approach* [26,25,30], one computes the behavior of a connector as a whole, implements this behavior, and runs this implementation sequentially as a centralized system. Which of those two approaches to choose may depend on the hardware architecture on which to deploy the application. For example, in the case of a service-oriented choreography application, the distributed approach seems natural, because the services involved run on different machines and the network between them may play a role in their coordination. However, if coordination involves computation threads running on the same machine in some multithreading application, the centralized approach appears more appropriate, because it avoids communication among the constituents of a connector at run-time: in this approach, due to the computation of the behavior of an entire connector at compile time, one abstracts from the individual, smaller, concurrent constituents of a connector to obtain one big sequential program for the whole (which can run in its own dedicated thread at run-time, among the computation threads it coordinates).

One optimization technique applicable to both the distributed and the centralized approaches involves the identification of the *synchronous* and the *asynchronous regions* of a connector [44]. A synchronous region contains exactly those nodes and channels of a connector that synchronize collectively to decide on their individual behavior; an asynchronous region connects synchronous regions in an asynchronous way, typically involving a fifo$_1$ channel. For instance, the connector consisting of a sync channel, a fifo$_1$ channel, and another sync channel (see Fig. 1d) has two synchronous regions, connected by an asynchronous region.

Intuitively, two synchronous regions can run completely independently of each other. Otherwise, by definition, those two subconnectors do not qualify as separate synchronous regions (instead, they constitute the same synchronous region). In the distributed approach, this means that nodes and channels need to share information only with those nodes and channels in the same synchronous region—not with every node or channel in the connector [44]. In the centralized approach, this means that one does not need to compute the behavior of a connector as a whole, but rather on a per-region basis [25]. Supplementary, asynchronous regions connect synchronous regions to each other by transporting data and control information between them. Based on how asynchronous regions do this, one can distinguish different versions of the region-based optimization technique, with different guarantees and for different use cases. For example, an asynchronous region can transport control information *directly* (in which case transportation starts at the same time as the coordination step that triggered it and ends before the next), *atomically* (same as the previous case but transportation can start also after the coordination step that triggered it), or *interleaved* (same as the previous case but transportation does not need to end before the next coordination step). Recent work shows that the region-based optimization technique for Reo can significantly improve performance [15,30,43,44] (both at compile time and at run-time), to the extent that its use will become vital for real-world applications: without it, automatically deploying (including code generation) and running connectors quickly becomes infeasible as their size increases.

Problem The region-based optimization technique still has a serious problem: although we have reason to *believe* (based on intuition and loose informal reasoning) that it preserves the semantics of a connector, we do not *know* this for sure by lack of a formal proof. In fact, in [15], Clarke and Proença identify one implementation of the region-based optimization technique that produces incorrect behavior for a certain class of connectors. An optimization as important as the region-based optimization technique for Reo should have a formal proof of correctness. The problem addressed in this paper is that such a proof currently does not exist.

Contributions of the paper In this paper, using the existing process algebraic semantics of Reo [35,32–34], we prove the correctness of the region-based optimization technique for asynchronous regions with direct transportation.[1] In this semantics, expressed using the specification language mCRL2 [20,22], one associates every connector with a process describing

[1] In practice, an implementation of the direct transportation version requires some form of synchronization between the different sides of an asynchronous region. On shared memory architectures, one can implement such synchronization relatively cheaply. On distributed memory architectures with

its behavior. Roughly, our proof technique consists of the formulation of a number of theorems for the untimed subset of mCRL2. We then apply these theorems to Reo's process algebraic semantics to prove the region-based optimization technique correct.

Importantly, however, the scope of this paper extends beyond Reo. Because we work on the semantics level—in terms of process algebra—and because we formulate our proof technique for general processes (not just those used in Reo's semantics), our results apply not exclusively to Reo but, instead, to any process in untimed mCRL2. As a result, we can divide the contributions of this paper into two categories: those concerning mCRL2 in general and those concerning Reo. More concretely:

- *mCRL2*
  - We define a *splitting procedure* for the untimed subset of mCRL2 and prove its correctness. Essentially, this procedure syntactically splits a process into two new processes: one process contains only actions from some set A; the other contains only actions from outside A.
  - Our work shows the feasibility of using the language mCRL2 (not the associated toolset) for proving properties of a whole language, Reo, rather than of individual concrete connectors. This subtly, yet significantly, differs from work of Kokash et al. [35,32–34], who introduced a process algebraic semantics of Reo for verifying concrete connectors (e.g., "this connector never deadlocks") but obtain no results about Reo as a language. As such, the work presented in this paper also paves the way to proving other properties about Reo using process algebra, including the correctness of others versions of the region-based optimization techniques (in terms of new different splitting procedures).
- *Reo*
  - We formalize the notion of (a)synchronous regions in terms of the process algebraic semantics of Reo.
  - We apply the splitting procedure to the process algebraic semantics of Reo, thereby justifying the region-based optimization technique for Reo implementations. To illustrate this further, we discuss how to implement and use the splitting procedure in the distributed approach, exploiting the local concurrency available on the computational nodes.

Although motivated by Reo, to emphasize the generality of our splitting procedure and theorems, we have organized the rest of this paper from a process algebra perspective; Reo serves as a 'case study' exemplifying their usefulness. In Section 2, we give an overview of the untimed subset of mCRL2 we use. In Section 3, to show mCRL2 in action, we summarize the process algebraic semantics of Reo. In Section 4, we introduce our splitting procedure, and in Section 5, we prove its correctness. In Section 6, we apply our splitting procedure to Reo. Section 7 contains related work, and Section 8 ends this paper with a conclusion and future work.

An earlier version of this work appeared in [27], where we considered the untimed data-free subset of mCRL2 and adopted a limited form of recursion. In this paper, by contrast, we do have data and a more general treatment of recursion. As a consequence, in addition to new proofs for new results, we necessarily revised, extended, and sometimes simplified many of our old proofs. The main challenge in doing so was extending the mechanism for "tracking choices"—crucial for our approach to work—to handle summation over data (with as few notational machinery as possible), which we did not have to deal with in [27]. Thus, although the idea behind our approach remained the same, we extended it and worked out the details under those new circumstances. Orthogonally, we realized that by using a transformation by Usenko that collapses a number of recursive process definitions into a single one (using mCRL2's data-dependent operators, including summation) [47], we could generalize our treatment of recursion from single recursive process definitions to systems of such definitions.

2. A process algebra with multiactions and data

The process algebra used in this paper is the untimed subset of mCRL2 [20,22], a specification language based on ACP [7] and the basis of the process algebraic semantics of Reo. Among other useful constructs, mCRL2 has one feature that makes it particularly well-suited as a semantic formalism for Reo, namely *multiactions*: collections of actions that occur at the same time. We postpone an explanation of how to use multiactions for describing the behavior of connectors until Section 3. In this section, we summarize the untimed subset of mCRL2.

2.1. Data

Before discussing the syntax and semantics of processes, we first give a terse overview of the data language of mCRL2, used to parameterize actions in the algebra (details appear elsewhere [20]). This data language, based on higher-order

substantial network delays, in contrast, the overhead of such synchronization may be prohibitively large. Therefore, the main application area of the work presented in this paper is shared memory implementations of Reo. Currently, all implementations of Reo can run on shared memory multithreading machines, including those following the distributed implementation approach. ("Distribution" in the distributed implementation approach refers to how the execution of connector constituents is divided over different execution units, irrespective of the underlying memory layout.)

$$
\begin{array}{ll}
\textbf{sort} & \mathbb{B} \\
\textbf{cons} & \mathit{true} : \mathbb{B},\ \mathit{false} : \mathbb{B} \\
\textbf{map} & \neg : \mathbb{B} \to \mathbb{B},\ \wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B},\ \ldots \\
\textbf{var} & b \\
\textbf{eqn} & \neg \mathit{true} = \mathit{false},\ \neg \mathit{false} = \mathit{true},\ \neg\neg b = b, \\
 & b \wedge \mathit{true} = b,\ b \wedge \mathit{false} = \mathit{false},\ \mathit{true} \wedge b = b,\ \mathit{false} \wedge b = \mathit{false},\ \ldots
\end{array}
$$

Fig. 2. Partial definition of sort $\mathbb{B}$.

$$
\begin{array}{lll}
a & ::= & \text{any action in } \mathbb{A}\text{ct} \\
\alpha & ::= & a(\mathbf{d}) \mid \tau \mid \alpha \sqcup \alpha \\
\boldsymbol{\alpha} & ::= & \alpha \mid \delta
\end{array}
$$

(a) Multiactions and deadlock.

$$
\begin{array}{lll}
p & ::= & \boldsymbol{\alpha} \mid P(\mathbf{d}) \mid p + p \mid p \cdot p \mid c \mathrel{\rightarrow} p \diamond p \mid \sum_{d \in D} p \\
 & & \mid p \parallel p \mid p \between q \mid p \mid p \\
 & & \mid \nabla_V(p) \mid \partial_B(p) \mid \rho_R(p) \mid \Gamma_C(p) \mid \mathcal{T}_I(p)
\end{array}
$$

(b) Processes.

Fig. 3. Syntax.

abstract data types, allows for the definition of *sorts*. Every sort consists of *constructors* and *maps*, which compose into *data expressions*. Every data expression can be interpreted as a *data element* of a sort. *Equations*, possibly containing *data variables* (over data expressions), enable one to derive equalities between data expressions (by giving meaning to maps). For example, Fig. 2 shows a fragment of the definition of mCRL2's built-in sort $\mathbb{B}$ [22], which represents the booleans. Additionally, mCRL2's collection of built-in sorts includes the natural numbers ($\mathbb{N}$) and the real numbers ($\mathbb{R}$). Users of mCRL2 can also define their own sorts.

Every sort S has, among other standard maps, a map $\approx : S \times S \to \mathbb{B}$ for equality of data expressions of sort S. For the built-in sorts, this map behaves as expected. For user-defined sorts, the user must provide equations that give meaning to $\approx$.

Henceforth, let c range over data expressions of sort $\mathbb{B}$, let d, e, f range over arbitrary data expressions, and let D, E, F range over such sets. Likewise, let $\mathbf{d}$, $\mathbf{e}$, $\mathbf{f}$ range over tuples of data expressions and data variables, and let $\mathbf{D}$, $\mathbf{E}$, $\mathbf{F}$ range over tuples of such sets. Finally, let x, y, z range over data variables, let X, Y, Z range over such sets, and let $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ range over such tuples. Furthermore:

Definition 1. $\mathbb{E}\text{lem}$ denotes a global set of data elements and $\mathbb{V}\text{ar}$ denotes a global set of data variables such that $\mathbb{E}\text{lem} \cap \mathbb{V}\text{ar} = \emptyset$.

2.2. Syntax

Fig. 3a shows the syntax of multiactions and deadlock.[2] Let $\mathbb{A}\text{ct}$ denote a global set of actions, ranged over by a, b, c (henceforth, whether c denotes an action or a data expression of sort $\mathbb{B}$ is always clear from the context). Actions can involve data, specified using the data language from Section 2.1. Note that data variables can occur in the parameter of $a(\mathbf{d})$. The distinguished symbol τ denotes the empty multiaction, which consists of no observable actions. Operator $\sqcup$ (associative and commutative) composes multiactions into larger multiactions; let $\mathbb{MA}\text{ct}$ denote the global set of all multiactions, ranged over by α, β, γ. The distinguished symbol δ denotes the deadlock process, which performs no multiactions; let $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ range over the processes in the set $\mathbb{MA}\text{ct} \cup \{\delta\}$.

Fig. 3b shows the syntax of processes. Parameterized *process references*, ranged over by $P(\mathbf{d})$, $Q(\mathbf{e})$, $R(\mathbf{f})$, refer to process definitions of the form $P(\mathbf{x} : \mathbf{D}) = p$, where p denotes some process: the process reference $P(\mathbf{d})$ behaves as the process resulting from substituting the occurrences of the data variables $\mathbf{x}$ with the data expressions $\mathbf{d}$ in p, denoted by $p[\mathbf{d}/\mathbf{x}]$. Processes, ranged over by p, q, r, consist of multiactions and process references, composed with a variety of operators as follows.

Basic operators Operator $+$ and $\cdot$ denote alternative and sequential composition in the usual way. Ternary operator $\_ \mathrel{\rightarrow} \_ \diamond \_$ composes processes into a conditional choice: the process $c \mathrel{\rightarrow} q \diamond r$ behaves as q if the data expression c equals *true* (in terms of $\approx$) and as r otherwise. Operator $\sum$ binds, for each data element in a *finite* set, a data variable in a process to that particular element and places the resulting processes in an alternative composition: the process $\sum_{x \in \{d_1, \ldots, d_\ell\}} q$, with $x \in \mathbb{V}\text{ar}$ and $d_1, \ldots, d_\ell \in \mathbb{E}\text{lem}$, behaves as $q[d_1/x] + \cdots + q[d_\ell/x]$ (shortly, we shall state this more explicitly in a proposition).[3]

Let Basic (defined in Fig. 5) denote the set of *basic processes*, which consist of only multiactions and the basic operators such that nested occurrences of $\sum$ bind different data variables. The latter restriction, imposed for technical convenience, does not really limit the expressiveness of the algebra, because one can always bring a process to the desired format by applying alpha-conversion (i.e., we consider processes up to alpha-conversion for summation). Furthermore, we associate

[2] The vertical bar "|" in the production rule of $\boldsymbol{\alpha}$ is a separator symbol; it does not denote the synchronous composition operator of the process algebra.

[3] In contrast to existing literature on mCRL2 [20], we define summation only for finite sets of data elements, because we prove all results in this paper only for finite summation: proving our main results for summation over infinite domains is still an open problem.

$$\begin{aligned} \mathsf{Bound}(\boldsymbol{\alpha}) &= \emptyset \\ \mathsf{Bound}(q+r)\,,\ \mathsf{Bound}(q\cdot r)\,,\ \mathsf{Bound}(c \twoheadrightarrow q \diamond r) &= \mathsf{Bound}(q) \cup \mathsf{Bound}(r) \\ \mathsf{Bound}(\textstyle\sum_{x\in D} q) &= \mathsf{Bound}(q) \cup \{x\} \end{aligned}$$

Fig. 4. Definition of Bound.

$$\begin{aligned} \boldsymbol{\alpha} &\in \mathsf{Basic} \\ q+r\,,\ q\cdot r\,,\ c \twoheadrightarrow q \diamond r &\in \mathsf{Basic} \ \textbf{iff}\ q\,,\ r \in \mathsf{Basic} \\ \textstyle\sum_{x\in D} q &\in \mathsf{Basic} \ \textbf{iff}\ \left[q \in \mathsf{Basic}\ \textbf{and}\ x \notin \mathsf{Bound}(q)\right] \end{aligned}$$

Fig. 5. Definition of Basic.

with every process p built from the operators discussed so far a set $\mathsf{Bound}(p)$ (defined in Fig. 4), which contains the data variables bound by occurrences of $\sum$ in p.

(Full mCRL2 contains also the *at* basic operator and the *initialization* basic operator for expressing timed behavior. We skip those operators here, because we use only the untimed subset of mCRL2 in this paper.)

Parallel operators Operator $\parallel$ interleaves and synchronizes processes. Operator $\between$ behaves as $\parallel$, but the first computation step must come from its left-hand argument. Similarly, operator $|$ behaves as $\parallel$, but the first computation step is formed by synchronizing the first multiaction of each of its arguments.

Additional operators Four additional operators constrain the behavior of processes composed in parallel. Operator ∇ restricts a process p to the multiactions in a set of nonempty multiactions $V \subseteq \mathbb{M}\mathbb{A}\mathrm{ct} \setminus \{\tau\}$ (modulo commutativity and associativity of $\sqcup$). Operator ∂ blocks those actions in a process p that occur also in a set of actions $B \subseteq \mathbb{A}\mathrm{ct}$. Operator ρ renames the actions in a process p according to a set of renaming rules $R \subseteq \mathbb{A}\mathrm{ct} \times \mathbb{A}\mathrm{ct}$. Finally, operator Γ applies the communication rules in a set $C \subseteq \mathbb{M}\mathbb{A}\mathrm{ct} \times \mathbb{A}\mathrm{ct}$ to a process p. We write communication rules as $\alpha \to a$ and require that τ does not occur in α.

Abstraction operator Operator $\mathcal{T}$ hides those actions in a process p that occur also in a set of actions $I \subseteq \mathbb{A}\mathrm{ct}$. The act of hiding an action a, which means "replacing a by τ," differs from the act of blocking a, which means "replacing a by δ."

We adopt the following usual operator precedence (in decreasing order): $\sqcup$, $|$, $\cdot$, $\parallel$, $\between$, $+$. We write as few parentheses as possible, omitting them also in the case of associative or commutative operators. For example, we write $p \cdot q \cdot r + \alpha + \beta$ instead of $(p \cdot (q \cdot r)) + (\alpha + \beta)$. Furthermore, let symbol $\oplus$ range over the binary operators $+$, $\cdot$, $\parallel$, $\between$, and $|$. Similarly, let symbol f range over unary operators ∇, ∂, ρ, Γ, and $\mathcal{T}$.

2.3. Semantics

Every process has an associated transition system describing its semantics (sos rules appear in [20]). Let $\simeq$ denote provable equality of processes. Fig. 6 shows a sound axiomatization for strong bisimulation of the operators shown in Fig. 3 [20].[4] Let function Free, which occurs in axioms SUM1, SUM2, and SUM5, map processes to the free data variables occurring in them. Note that Fig. 6 axiomatizes three additional operators on multiactions: operator $\setminus$ subtracts the multiaction on its right-hand side from the multiaction on its left-hand side; operator $\sqsubseteq$ checks if the multiaction on its right-hand side contains the multiaction on its left-hand side; operator $\_$ clears a multiaction from data parameters. These three additional operators occur in the definition of the auxiliary function $\mathcal{C}$, used in Axiom C1:

$$\mathcal{C}_C(\alpha) = \begin{cases} \mathcal{C}_{C_1}(\mathcal{C}_{C_2}(\alpha)) & \textbf{if}\ C = C_1 \cup C_2\ \textbf{and}\ C_1 \cap C_2 = \emptyset\ \textbf{and}\ C_1\,,\ C_2 \neq \emptyset \\ b(\mathbf{e}) \sqcup \mathcal{C}_C(\alpha \setminus \beta) & \textbf{if}\ C = \{\underline{\beta} \to b\}\ \textbf{and}\ \beta = b_1(\mathbf{e}) \sqcup \cdots \sqcup b_m(\mathbf{e})\ \textbf{and}\ \beta \sqsubseteq \alpha \\ \alpha & \textbf{otherwise} \end{cases}$$

Informally, $\mathcal{C}$ applies the communication rules in a set C to a multiaction α. The left-hand sides of different communication rules in C must be pairwise disjoint [20]. Moreover, an action may not occur both in a left-hand side and in a right-hand side [22]. These constraints ensure that $\mathcal{C}_{C_1}(\mathcal{C}_{C_2}(\alpha)) = \mathcal{C}_{C_2}(\mathcal{C}_{C_1}(\alpha))$ holds, which guarantees that $\mathcal{C}_{C_1 \cup C_2}$ has a unique solution [20]. For instance, $C = \{a \sqcup b \to b\,,\ a \sqcup c \to c\}$ is forbidden, because a occurs in both left-hand sides.

Although we use only a subset of the axioms in Fig. 6 in proofs, we show all of them for completeness.[5]

The proof of one of the theorems in Section 5 relies on the *recursive specification principle* (RSP) [8]. This principle states that every *guarded* recursive definition has at most one solution.[6] One can formulate this principle in terms of a guarded *process operator* Φ—a function from processes to processes—as follows [22, Section 9.6]:

$$P \simeq \Phi(P)\ \textbf{and}\ Q \simeq \Phi(Q)\ \textbf{implies}\ P \simeq Q$$

Thus, if Φ has both P and Q as fixed points, P must be provably equal to Q.

Finally, we introduce a "metalevel" operator $\bigsqcup$ to abbreviate arbitrary finite sequences of multiactions composed together: let $\bigsqcup_{i=1}^{n} \alpha_i$ abbreviate the multiaction $\alpha_1 \sqcup \cdots \sqcup \alpha_n$ (if $n > 0$) or τ (if $n = 0$). Similarly, we introduce a metalevel operator $\sum$ (same symbol as the summation operator but with a different, yet related, meaning) to abbreviate alternative

[4] The axiomatization is also *relatively complete* for processes without process references. This means that completeness of the process language depends on completeness of the data language [20].

[5] We use the following axioms: MA2, MA3, A4, A6, A7, COND1, COND2, M, LM1, LM2, LM3, LM4, S3, S4, S5, S6, S7, SMA, B1, B2, B3, B4, C1, H1, H2, H3, H4, F1, F2, F3, F4. Furthermore, we assume that summations have been alpha-converted to the desired format by SUM2.

[6] In a guarded recursive definition $P = p$, every occurrence of P in p is *preceded* by a multiaction, where "precedence" is defined as follows [8]: if q' is a subprocess of q (e.g., a multiaction) and r' is a subprocess of r (e.g., a process reference), q' precedes r' in $q \cdot r$.

MA1	$\alpha \sqcup \beta \simeq \beta \sqcup \alpha$	M	$p \parallel q \simeq p \between q + q \between p + p \mid q$
MA2	$(\alpha \sqcup \beta) \sqcup \gamma \simeq \alpha \sqcup (\beta \sqcup \gamma)$		
MA3	$\alpha \sqcup \tau \simeq \alpha$	LM1	$\boldsymbol{\alpha} \between p \simeq \boldsymbol{\alpha} \cdot p$
		LM2	$\delta \between p \simeq \delta$
SMA	$\alpha \mid \beta \simeq \alpha \sqcup \beta$	LM3	$\boldsymbol{\alpha} \cdot p \between q \simeq \boldsymbol{\alpha} \cdot (p \parallel q)$
		LM4	$(p + q) \between r \simeq p \between r + q \between r$
MD1	$\tau \setminus \alpha \simeq \tau$		
MD2	$\alpha \setminus \tau \simeq \alpha$	S1	$p \mid q \simeq q \mid p$
MD3	$\alpha \setminus (\beta \sqcup \gamma) \simeq (\alpha \setminus \beta) \setminus \gamma$	S2	$(p \mid q) \mid r \simeq p \mid (q \mid r)$
MD4	$(a(\mathbf{d}) \sqcup \alpha) \setminus a(\mathbf{d}) \simeq \alpha$	S3	$p \mid \tau \simeq p$
MD5	$(a(\mathbf{d}) \sqcup \alpha) \setminus b(\mathbf{e}) \simeq a(\mathbf{d}) \sqcup (\alpha \setminus b(\mathbf{e}))$	S4	$\boldsymbol{\alpha} \mid \delta \simeq \delta$
	if $[a \neq b$ **or** $\mathbf{d} \neq \mathbf{e}]$	S5	$(\boldsymbol{\alpha} \cdot p) \mid \boldsymbol{\beta} \simeq \boldsymbol{\alpha} \mid \boldsymbol{\beta} \cdot p$
		S6	$(\boldsymbol{\alpha} \cdot p) \mid (\boldsymbol{\beta} \cdot q) \simeq \boldsymbol{\alpha} \mid \boldsymbol{\beta} \cdot (p \parallel q)$
MS1	$\tau \sqsubseteq \alpha \simeq \mathit{true}$	S7	$(p + q) \mid r \simeq p \mid r + q \mid r$
MS2	$a(\mathbf{d}) \sqsubseteq \tau \simeq \mathit{false}$		
MS3	$a(\mathbf{d}) \sqcup \alpha \sqsubseteq a(\mathbf{d}) \sqcup \beta \simeq \alpha \sqsubseteq \beta$	V1	$\nabla_V(\alpha) \simeq \alpha$ **if** $\underline{\alpha} \in V \cup \{\tau\}$
MS4	$a(\mathbf{d}) \sqcup \alpha \sqsubseteq b(\mathbf{e}) \sqcup \beta \simeq a(\mathbf{d}) \sqcup (\alpha \setminus b(\mathbf{e})) \sqsubseteq \beta$	V2	$\nabla_V(\alpha) \simeq \delta$ **if** $\underline{\alpha} \notin V \cup \{\tau\}$
	if $[a \neq b$ **or** $\mathbf{d} \neq \mathbf{e}]$		
		B1	$\partial_B(\tau) \simeq \tau$
MAN1	$\underline{\tau} \simeq \tau$	B2	$\partial_B(a(\mathbf{d})) \simeq a(\mathbf{d})$ **if** $a \notin B$
MAN2	$\underline{a(\mathbf{d})} \simeq a$	B3	$\partial_B(a(\mathbf{d})) \simeq \delta$ **if** $a \in B$
MAN3	$\underline{\alpha \sqcup \beta} \simeq \underline{\alpha} \sqcup \underline{\beta}$	B4	$\partial_B(\alpha \mid \beta) \simeq \partial_B(\alpha) \mid \partial_B(\beta)$
A1	$p + q \simeq q + p$	R1	$\rho_R(\tau) \simeq \tau$
A2	$p + (q + r) \simeq (p + q) + r$	R2	$\rho_R(a(\mathbf{d})) \simeq b(\mathbf{d})$
A3	$p + p \simeq p$		**if** $a \to b \in R$ **for some** b
A4	$(p + q) \cdot r \simeq p \cdot r + q \cdot r$	R3	$\rho_R(a(\mathbf{d})) \simeq a(\mathbf{d})$
A5	$(p \cdot q) \cdot r \simeq p \cdot (q \cdot r)$		**if** $a \to b \notin R$ **for all** b
A6	$p + \delta \simeq p$	R4	$\rho_R(\alpha \mid \beta) \simeq \rho_R(\alpha) \mid \rho_R(\beta)$
A7	$\delta \cdot p \simeq \delta$		
		C1	$\Gamma_C(\alpha) \simeq \mathcal{C}_C(\alpha)$
COND1	$\mathit{true} \rightarrowtriangle p \diamond q \simeq p$		
COND2	$\mathit{false} \rightarrowtriangle p \diamond q \simeq q$	H1	$\mathcal{T}_I(\tau) \simeq \tau$
		H2	$\mathcal{T}_I(a(\mathbf{d})) \simeq \tau$ **if** $a \in I$
SUM1	$\sum_{x \in D} p \simeq p$ **if** $x \notin \mathsf{Free}(p)$	H3	$\mathcal{T}_I(a(\mathbf{d})) \simeq a(\mathbf{d})$ **if** $a \notin I$
SUM2	$\sum_{x \in D} p \simeq \sum_{y \in D} p[y/x]$	H4	$\mathcal{T}_I(\alpha \mid \beta) \simeq \mathcal{T}_I(\alpha) \mid \mathcal{T}_I(\beta)$
	if $y \notin \mathsf{Free}(p)$		
SUM3	$\sum_{x \in D} p \simeq \sum_{x \in D} p + p$	F1	$f(\delta) \simeq \delta$
SUM4	$\sum_{x \in D} (p + q) \simeq \sum_{x \in D} p + \sum_{x \in D} q$	F2	$f(\alpha + \beta) \simeq f(\alpha) + f(\beta)$
SUM5	$(\sum_{x \in D} p) \cdot q \simeq \sum_{x \in D} (p \cdot q)$	F3	$f(\alpha \cdot \beta) \simeq f(\alpha) \cdot f(\beta)$
	if $x \notin \mathsf{Free}(q)$	F4	$f(\sum_{x \in D} p) \simeq \sum_{x \in D} f(p)$

Fig. 6. Axioms.

compositions consisting of a finite number of processes: let $\sum_{i=1}^{n} p_i$ abbreviate the process $p_1 + \cdots + p_n$ (if $n > 0$) or δ (if $n = 0$). Operators $\between$ and $\sum$ help us in formulating propositions and proofs more concisely. Although strictly different, the latter has a tight connection with the summation operator. The following proposition makes this connection precise.

Proposition 1. *(See [22, Section 4.6].)* $\sum_{x \in \{d_1, \ldots, d_\ell\}} q \simeq \sum_{i=1}^{l} q[d_i/x]$.

3. An application of the algebra: semantics of Reo

Before continuing with our splitting procedure in Section 4, we briefly discuss Reo and its process algebraic semantics by Kokash et al. [35,32–34] as an application of the algebra discussed in Section 2; this also helps us to relate the relatively abstract discussion in Section 4 to a concrete case. Recall from Section 1 that connectors consist of channels and nodes. Below, following Kokash et al., we outline how these channels and nodes, as well as the data they transport, behave and how to describe such behavior as processes in untimed mCRL2.

Data To model the pieces of data transported by a connector in mCRL2, one can define a sort whose constructors correspond to concrete data items. Additionally, one can define maps to allow channels to perform operations on data elements, but we skip that here. Let $\mathbb{D}\mathsf{ata} \subseteq \mathbb{E}\mathsf{lem}$ denote a finite global set of data elements of said sort.

Channels Every channel has exactly two ends, each of which has one of two types: *source ends* accept data, while *sink ends* dispense data. Besides this assumption on the number of ends, Reo makes no assumptions about channels. This means, for example, that Reo allows channels with two source ends. Fig. 7 shows the graphical syntax of four common channels, a textual syntax, and an informal description of their behavior.

In the process algebraic semantics of Reo, one associates every channel end with an action. For source ends, such an action represents the acceptance of data; for sink ends, it represents the dispersal of data. By composing these actions into

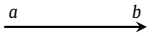
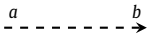
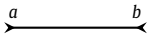
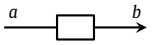
Graphical syntax	*Textual syntax*	*Semantics*
	sync⟨*a*; *b*⟩	Atomically accepts an item on its source end *a* and dispenses it on its sink end *b*.
	lossysync⟨*a*; *b*⟩	Atomically accepts an item on its source end *a* and, nondeterministically, either dispenses it on its sink end *b* or loses it.
	syncdrain⟨*a*, *b*; ⟩	Atomically accepts (and loses) items on both of its source ends *a* and *b*.
	fifo1⟨*a*; *b*⟩	Atomically accepts an item on its source end and stores it in its buffer, then atomically dispenses the item *d* on its sink end and clears its buffer.

Fig. 7. Syntax and informal semantics of common channels.

multiactions, one can describe channels that atomically accept and dispense data on their ends. For example, the following process definitions describe the behavior of the channels in Fig. 7.[7]

$$\begin{aligned}
\texttt{Sync}\langle \texttt{a}; \texttt{b}\rangle &= \textstyle\sum_{x\in\mathbb{D}\text{ata}} \texttt{a}(x) \sqcup \texttt{b}(x) \cdot \texttt{Sync}\langle \texttt{a}; \texttt{b}\rangle \\
\texttt{LossySync}\langle \texttt{a}; \texttt{b}\rangle &= \textstyle\sum_{x\in\mathbb{D}\text{ata}} (\texttt{a}(x) \sqcup \texttt{b}(x) + \texttt{a}(x)) \cdot \texttt{LossySync}\langle \texttt{a}; \texttt{b}\rangle \\
\texttt{SyncDrain}\langle \texttt{a}, \texttt{b};\rangle &= \textstyle\sum_{x\in\mathbb{D}\text{ata}} \sum_{y\in\mathbb{D}\text{ata}} \texttt{a}(x) \sqcup \texttt{b}(y) \cdot \texttt{SyncDrain}\langle \texttt{a}, \texttt{b};\rangle \\
\texttt{Fifo1}\langle \texttt{a}; \texttt{b}\rangle &= \textstyle\sum_{x\in\mathbb{D}\text{ata}} (\texttt{a}(x) \cdot \texttt{b}(x)) \cdot \texttt{Fifo1}\langle \texttt{a}; \texttt{b}\rangle
\end{aligned}$$

The definition of $\texttt{Sync}\langle \texttt{a}; \texttt{b}\rangle$ models synchronous flow of a data item x through channel ends a and b, represented by the multiaction $\texttt{a}(x) \sqcup \texttt{b}(x)$. The definition of $\texttt{LossySync}\langle \texttt{a}; \texttt{b}\rangle$ models a (nondeterministic) choice between flow of a data item x through ends a and b and flow of x through only a, represented by the process $\texttt{a}(x) \sqcup \texttt{b}(x) + \texttt{a}(x)$. The definition of $\texttt{SyncDrain}\langle \texttt{a}, \texttt{b};\rangle$ models synchronous flow of (unrelated) data items x and y through channel ends a and b, represented by the multiaction $\texttt{a}(x) \sqcup \texttt{b}(y)$. The definition of $\texttt{Fifo1}\langle \texttt{a}; \texttt{b}\rangle$ models flow of a data item x through channel end a followed by flow of the same x through channel end b. The recursion present in each of the four process definitions above models that the channels repeat their behavior indefinitely.

In this paper, we adopt the context-*in*sensitive process algebraic semantics of Reo, originally based on *constraint automata* [5]. In context-insensitive semantic formalisms, one cannot directly describe channels and connectors whose behavior depends not only on their internal state but also on the presence or absence of I/O operations—their *context*. In contrast, one can describe such channels and connectors in semantic formalisms that do support context-sensitivity. For instance, a context-sensitive version of lossysync should lose a data item only in the absence of I/O operations on its sink end. A context-sensitive process algebraic semantics of Reo exists [34,35], originally based on *connector coloring* with three colors [14]. Alternatively, we could *encode* a context-sensitive process algebraic semantics along the lines of [29].[8] Although the splitting procedure introduced in Section 4 supports both approaches, we do not pursue context sensitivity in this paper, because it would only distract and unnecessarily complicate matters.

Nodes Entities communicating through a connector perform I/O operations—writes and takes—on its nodes. Reo features three kinds of nodes: *source nodes* on which only source ends coincide, *sink nodes* on which only sink ends coincide, and *mixed nodes* on which both kinds of channel end coincide. Nodes have the following semantics.

- A source node n has *replicator semantics*. Once an entity attempts to write a data item d on n, this node first suspends this operation. Subsequently, n notifies the channels whose source ends coincide on n that it offers d. Once each of these channels has notified n that it accepts d, n resolves the write: atomically, n dispenses d to each of its coincident source ends.
- A sink node n has *nondeterministic merger semantics*. Once an entity attempts to take a data item from n, this node first suspends this operation. Subsequently, n notifies the channels whose sink ends coincide on n that it accepts a data item. Once at least one of these channels has notified n that it offers a data item, n resolves the take: atomically, n fetches this data item from the appropriate channel end and dispenses it to the entity attempting to take. If multiple sink ends offer a data item, n chooses one of them nondeterministically.
- A mixed node n has *pumping station semantics*, which is a combination of the replicator semantics and merger semantics discussed above, where fetching and dispensing occur atomically.

In the process algebraic semantics of Reo, one associates each of the m source ends of a node with an action $\texttt{src}_i$ $(1 \leqslant i \leqslant m)$ and each of its n sink ends with an action $\texttt{snk}_i$ $(1 \leqslant i \leqslant n)$. Then, one can describe nodes by combining the

[7] In process references, in contrast to the textual syntax in Fig. 7, angle brackets have no meaning and give no structure.
[8] An extensive overview of context-(in)sensitive semantic formalisms for Reo appears in [24].

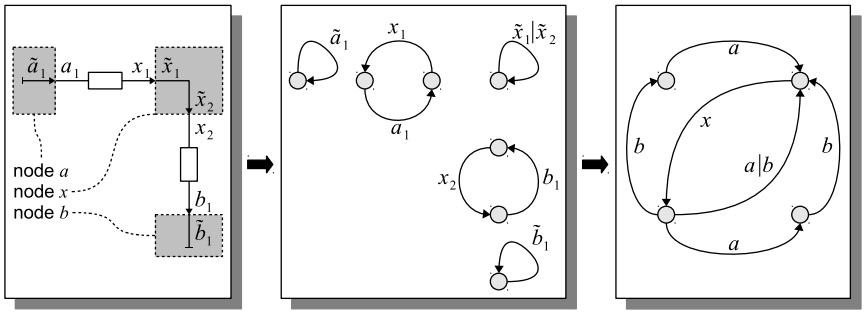


Fig. 8. Labeled transition system(s) of the process(es) modeling the connector in Fig. 1a. On the left is a graphical representation of the decomposition of that connector into channels and nodes with labeled ends. In the middle are the labeled transition systems of the processes modeling those channels and nodes (without data for simplicity). On the right is the labeled transition system of the parallel composition of those processes (after applying communication and blocking).

processes for a binary replicator R (one sink end to two source ends), a binary merger M (two sink ends to one source end), a one-to-one pumping station PS, and a boundary node B:

$$\begin{aligned}
\mathtt{R}\langle \mathtt{snk}; \mathtt{src}_1, \mathtt{src}_2\rangle &= \textstyle\sum_{x\in\mathbb{D}\mathrm{ata}} \mathtt{snk}(x) \sqcup \mathtt{src}_1(x) \sqcup \mathtt{src}_2(x) \cdot \mathtt{R}\langle \mathtt{snk}; \mathtt{src}_1, \mathtt{src}_2\rangle \\
\mathtt{M}\langle \mathtt{snk}_1, \mathtt{snk}_2; \mathtt{src}\rangle &= \textstyle\sum_{x\in\mathbb{D}\mathrm{ata}} (\mathtt{snk}_1(x) \sqcup \mathtt{src}(x) + \mathtt{snk}_2(x) \sqcup \mathtt{src}(x)) \cdot \mathtt{M}\langle \mathtt{snk}_1, \mathtt{snk}_2; \mathtt{src}\rangle \\
\langle \mathtt{snk}; \mathtt{src}\rangle &= \textstyle\sum_{x\in\mathbb{D}\mathrm{ata}} \mathtt{snk}(x) \sqcup \mathtt{src}(x) \cdot \langle \mathtt{snk}; \mathtt{src}\rangle \\
\mathtt{B}\langle \mathtt{bnd}\rangle &= \textstyle\sum_{x\in\mathbb{D}\mathrm{ata}} \mathtt{bnd}(x) \cdot \mathtt{B}\langle \mathtt{bnd}\rangle
\end{aligned}$$

Connectors To get the behavior of a connector as a process, one composes the processes of the constituents of that connector in parallel and synchronizes their actions. Below, we give the processes of the connectors in Figs. 1a and 1c. Fig. 8 additionally shows the labeled transition system(s) of the process(es) modeling the connector in Fig. 1a. More examples may be found in [32–35].

$$\begin{aligned}
\textit{Fig. 1a} &= \partial_{\{\mathtt{a}_1,\widetilde{\mathtt{a}}_1,\mathtt{x}_1,\widetilde{\mathtt{x}}_1,\mathtt{x}_2,\widetilde{\mathtt{x}}_2,\mathtt{b}_1,\widetilde{\mathtt{b}}_1\}}(\Gamma_{\{\mathtt{a}_1\sqcup\widetilde{\mathtt{a}}_1\to\mathtt{a},\mathtt{x}_1\sqcup\widetilde{\mathtt{x}}_1\sqcup\mathtt{x}_2\sqcup\widetilde{\mathtt{x}}_2\to\mathtt{x},\mathtt{b}_1\sqcup\widetilde{\mathtt{b}}_1\to\mathtt{b}\}}(q)) \\
\textit{Fig. 1c} &= \partial_{\{a_i,\widetilde{a}_i \mid a\in\{\mathtt{a},\mathtt{b},\mathtt{c}\}\wedge i\in\{1,2,3\}\}}(\Gamma_{\{a_1\sqcup\widetilde{a}_1\sqcup a_2\sqcup\widetilde{a}_2\sqcup a_3\sqcup\widetilde{a}_3\to a \mid a\in\{\mathtt{a},\mathtt{b},\mathtt{c}\}\}}(r))
\end{aligned}$$

For:

$$q = \begin{pmatrix} \mathtt{B}\langle\widetilde{\mathtt{a}}_1\rangle \parallel \mathtt{Fifo1}\langle\mathtt{a}_1;\mathtt{x}_1\rangle \parallel \langle\widetilde{\mathtt{x}}_1;\widetilde{\mathtt{x}}_2\rangle \\ \qquad\qquad\qquad\qquad \parallel \mathtt{Fifo1}\langle\mathtt{x}_2;\mathtt{b}_1\rangle \\ \qquad\qquad\qquad\qquad \parallel \mathtt{B}\langle\widetilde{\mathtt{b}}_1\rangle \end{pmatrix}$$

$$r = \begin{pmatrix} \mathtt{B}\langle\mathtt{a}_1\rangle \parallel \mathtt{R}\langle\widetilde{\mathtt{a}}_1;\widetilde{\mathtt{a}}_2,\widetilde{\mathtt{a}}_3\rangle \parallel \mathtt{Sync}\langle\mathtt{a}_3;\mathtt{c}_1\rangle \parallel \mathtt{M}\langle\widetilde{\mathtt{c}}_1,\widetilde{\mathtt{c}}_2;\widetilde{\mathtt{c}}_3\rangle \parallel \mathtt{B}\langle\mathtt{c}_3\rangle \\ \qquad \parallel \mathtt{SyncDrain}\langle\mathtt{b}_2,\mathtt{a}_2\rangle \parallel \mathtt{Fifo1}\langle\mathtt{b}_3;\mathtt{c}_2\rangle \parallel \\ \mathtt{B}\langle\mathtt{b}_1\rangle \parallel \mathtt{R}\langle\widetilde{\mathtt{b}}_1;\widetilde{\mathtt{b}}_2,\widetilde{\mathtt{b}}_3\rangle \end{pmatrix}$$

4. Splitting processes

Recall from Section 1 that we originally aimed at establishing the validity of optimizing implementations of Reo through the identification of (a)synchronous regions. Essentially, we want to show that splitting connectors along the boundaries of their (a)synchronous regions (and running the resulting subconnectors concurrently) neither loses behavior nor gives rise to inadmissible behavior. In this section, we lay the foundation for this kind of splitting in terms of a splitting procedure for processes. Later, in Section 6, we apply this procedure to the process algebraic semantics of Reo, thereby justifying the splitting of connectors. Here, in Section 4.1, we start by explaining the intuition behind our splitting procedure; formal definitions appear in Section 4.2. In Section 5, we investigate and prove properties of our splitting procedure, including a proof of correctness. We note that our notion of "splitting processes" differs from "uniquely decomposing processes" [40]: in our context, neither primality nor uniqueness of processes matters. We discuss the differences in more detail in Section 7.

4.1. Intuition

For simplicity, to convey the intuition behind our splitting procedure, we consider only data-free processes in this subsection (definitions in Section 4.2 do incorporate data).

$\mathsf{Act}(a(\mathbf{d}))$	$= \{a\}$
$\mathsf{Act}(\tau)$, $\mathsf{Act}(\delta)$, $P(\mathbf{d})$	$= \emptyset$
$\mathsf{Act}(\beta \sqcup \gamma)$	$= \mathsf{Act}(\beta) \cup \mathsf{Act}(\gamma)$
$\mathsf{Act}(q \oplus r)$, $\mathsf{Act}(c \twoheadrightarrow q \diamond r)$	$= \mathsf{Act}(q) \cup \mathsf{Act}(r)$
$\mathsf{Act}(\sum_{x \in D} q)$, $\mathsf{Act}(f(q))$	$= \mathsf{Act}(q)$

Fig. 9. Definition of Act.

Let $\mathsf{Act}(p)$ (defined in Fig. 9) denote the set of actions syntactically occurring in a process p.[9] We introduce function split, which splits a process p along a set of actions $\mathbb{A} \subseteq \mathbb{A}\mathsf{ct}$ into two processes: one of these processes contains no actions in $\mathsf{Act}(p) \setminus \mathbb{A}$, while the other process contains no actions in $\mathbb{A}$. We call the former process the $\mathbb{A}$-*isolation of* p and the latter process the $\mathbb{A}$-*coisolation of* p. We aim at constructing p's isolation and its coisolation such that their parallel composition behaves as p under some appropriate notion of synchronization (defined shortly).

Informally, to construct p's $\mathbb{A}$-isolation, replace every action in p as follows:

- If $a \in \mathbb{A}$, replace a with the multiaction $a \sqcup \xi(a)$, where $\xi(a)$ denotes a fresh *auxiliary action* with respect to $\mathsf{Act}(p)$. Intuitively, $\xi(a)$ represents the act of "disseminating that this process performs a."
- If $b \notin \mathbb{A}$, replace b with the auxiliary action $\overline{\xi}(b)$, where $\overline{\xi}(b)$ denotes a fresh action with respect to $\mathsf{Act}(p)$. Intuitively, $\overline{\xi}(b)$ represents the act of "discovering that another process performs b."

Symmetrically, to construct the $\mathbb{A}$-coisolation of a process p, replace in p every $b \in \mathbb{A}$ with $\overline{\xi}(b)$ and every $b \notin \mathbb{A}$ with $b \sqcup \xi(b)$. Note that because the foregoing affects only multiactions, p's isolation and its coisolation have the same syntactic structure as p. In other words: the process p, its isolation, and its coisolation have the same transition system modulo transition labels.

To illustrate isolation and coisolation, consider an example process $q = a \cdot b$. This process has $q_1 = a \sqcup \xi(a) \cdot \overline{\xi}(b)$ as its $\{a\}$-isolation and $q_2 = \overline{\xi}(a) \cdot b \sqcup \xi(b)$ as its $\{a\}$-coisolation. The parallel composition of q_1 and q_2, however, does not behave as q yet: to ensure that a process behaves as the parallel composition of its isolation and its coisolation, these two processes should appropriately synchronize on $\xi(a)$ and $\overline{\xi}(a)$ for each a. To this end, we apply the communication operator Γ to such compositions. In our example, this yields the process $\Gamma_C(q_1 \parallel q_2)$ for $C = \{\xi(a) \sqcup \overline{\xi}(a) \to \mathtt{tau}, \xi(b) \sqcup \overline{\xi}(b) \to \mathtt{tau}\}$. The special action `tau` serves as a placeholder action for τ, and we can hide it immediately using the abstraction operator $\mathcal{T}$[10]; henceforth, without loss of generality, we assume $\mathtt{tau} \notin \mathsf{Act}(p)$ for each p. In our example, this yields the process $\mathcal{T}_I(\Gamma_C(q_1 \parallel q_2))$ with $I = \{\mathtt{tau}\}$ and C as before.

But also this process does not behave as q yet: synchronization and abstraction alone do not suffice—we must also block those auxiliary actions whose individual performance "makes no sense." For instance, we consider every unpaired occurrence of $\overline{\xi}(a)$ in a multiaction α nonsensical: intuitively, performing $\overline{\xi}(a)$ suggests that some process discovers that another process performs a, even though this does not happen (otherwise, also $\xi(a)$ would occur in α). By symmetry, we consider also every unpaired occurrence of $\xi(a)$ nonsensical. To block unpaired occurrences of $\xi(a)$ and $\overline{\xi}(a)$, we apply the blocking operator ∂. In our example, this yields the process $\partial_B(\mathcal{T}_I(\Gamma_C(q_1 \parallel q_2)))$ with $B = \{\xi(a), \overline{\xi}(a), \xi(b), \overline{\xi}(b)\}$ and I and C as before. This process behaves as q, concluding our example.

We proceed with general formal definitions of the splitting procedure just outlined.

4.2. Formal definitions

Auxiliary actions and substitution environments We start with a formal account of the fresh auxiliary actions of the form $\xi(a)$ and $\overline{\xi}(a)$. As suggested by this notation, ξ and $\overline{\xi}$ denote functions that take an action a as input and produce another action as output. We collect such pairs of functions in *substitution environments* as follows. Let C^* denote the set of finite strings over C.

Definition 2. `1` and `2` are global symbols such that $\mathtt{1} \neq \mathtt{2}$ and $\mathtt{1}, \mathtt{2} \notin \mathbb{E}\mathsf{lem} \cup \mathbb{V}\mathsf{ar}$.

Definition 3. A substitution environment, typically denoted by Ξ, is a quadruple $\langle A, \mathtt{tau}, \xi, \overline{\xi} \rangle$ consisting of a set $A \subseteq \mathbb{A}\mathsf{ct}$, an action $\mathtt{tau} \in \mathbb{A}\mathsf{ct} \setminus A$ and injective functions $\xi, \overline{\xi} : \{\mathtt{1}, \mathtt{2}\}^* \times A \rightarrowtail \mathbb{A}\mathsf{ct} \setminus (A \cup \{\mathtt{tau}\})$ such that $\mathsf{img}(\xi) \cap \mathsf{img}(\overline{\xi}) = \emptyset$.

Example. Suppose:

- $\mathbb{A}\mathsf{ct} = \{\mathtt{tau}, \mathtt{foo}\} \cup \{\mathtt{x\_foo\_}w \mid w \in \{\mathtt{1}, \mathtt{2}\}^*\} \cup \{\overline{\mathtt{x\_foo\_}w} \mid w \in \{\mathtt{1}, \mathtt{2}\}^*\}$

[9] We use the formal definition of Act only to formulate a technical requirement in premises of propositions, lemmas, and theorems in Section 5. This requirement says something about the actions that syntactically occur in a process. Therefore, the definition of Act does not need to take the effect of ∇, ∂, ρ, Γ, and $\mathcal{T}$ into account. See also the example at the end of Section 4.2.

[10] We use this construction, because mCRL2 does not permit communications to map directly to τ.

$$
\begin{array}{ll}
\mathsf{dom}(\Xi) & = \{a \mid \langle w\,,\,a\rangle \in \mathrm{dom}(\xi) \cap \mathrm{dom}(\bar{\xi})\} \\
\mathsf{img}(\Xi) & = \mathrm{img}(\xi) \cup \mathrm{img}(\bar{\xi}) \\
\mathsf{comm}(\Xi) & = \{\xi_w(a) \sqcup \bar{\xi}_w(a) \to \mathtt{tau} \mid \langle w\,,\,a\rangle \in \mathrm{dom}(\xi) \cap \mathrm{dom}(\bar{\xi})\}
\end{array}
$$

Fig. 10. Definitions of dom, img, and comm.

- $A = \{\mathtt{foo}\}$
- $\xi = \{\langle w\,,\,\mathtt{foo}\rangle \mapsto \mathtt{x\_foo\_}w \mid w \in \{1\,,\,2\}^*\}$
- $\bar{\xi} = \{\langle w\,,\,\mathtt{foo}\rangle \mapsto \overline{\mathtt{x\_foo\_}w} \mid w \in \{1\,,\,2\}^*\}$

Then, $\langle A\,,\,\mathtt{tau}\,,\,\xi\,,\,\bar{\xi}\rangle$ is a substitution environment. □

Henceforth, we write $\xi_w(a)$ and $\bar{\xi}_w(a)$ instead of $\xi(w\,,\,a)$ and $\bar{\xi}(w\,,\,a)$. Note that we dropped the w subscripts in the example in Section 4.1: because we did not need such an extra string of information, we omitted it for simplicity. In the general case, however, this information plays a vital role, as explained shortly.

Let "dom" and "img" map functions to their domain and image. Fig. 10 shows auxiliary functions for substitution environment. Functions dom and img map substitution environments to their domain (projected on actions) and image. Function comm maps substitution environments to communications derivable from them.

Example. Suppose $\Xi = \langle A\,,\,\mathtt{tau}\,,\,\xi\,,\,\bar{\xi}\rangle$ is the substitution environment defined in the Example on page 55. Then:

- $\mathsf{dom}(\xi) = \mathrm{dom}(\bar{\xi}) = \{\langle w\,,\,\mathtt{foo}\rangle \mid w \in \{1\,,\,2\}^*\}$
- $\mathsf{img}(\xi) = \{\mathtt{x\_foo\_}w \mid w \in \{1\,,\,2\}^*\}$
- $\mathsf{img}(\bar{\xi}) = \{\overline{\mathtt{x\_foo\_}w} \mid w \in \{1\,,\,2\}^*\}$

And:

- $\mathsf{dom}(\Xi) = \{\mathtt{foo}\}$
- $\mathsf{img}(\Xi) = \bigcup_{w \in \{1\,,\,2\}^*} \{\mathtt{x\_foo\_}w\,,\,\overline{\mathtt{x\_foo\_}w}\}$
- $\mathsf{comm}(\Xi) = \{\mathtt{x\_foo\_}w \sqcup \overline{\mathtt{x\_foo\_}w} \to \mathtt{tau} \mid w \in \{1\,,\,2\}^*\}$ □

Henceforth, to avoid heavy notation, we quantify implicitly over all substitution environments in definitions, propositions, lemmas, theorems, and proofs, without mentioning them explicitly. We do the same for sets $\mathbb{A}$, which contain the actions along which we split processes.

Isolation and coisolation To formalize the notions of $\mathbb{A}$-isolation and $\mathbb{A}$-coisolation, we introduce the functions isol and $\overline{\mathsf{isol}}$, ranged over by $\widehat{\mathsf{isol}}$. Fig. 11 shows their definitions. (Recall that we quantify implicitly over all execution environments Ξ and sets $\mathbb{A}$ without mentioning them explicitly.) Functions isol and $\overline{\mathsf{isol}}$ take a string over $\{1\,,\,2\} \cup \mathbb{E}\mathsf{lem} \cup \mathbb{V}\mathsf{ar}$ and a basic process as input.[11]

Before we take a closer look at Fig. 11, we explain the purpose of the string over $\{1\,,\,2\} \cup \mathbb{E}\mathsf{lem} \cup \mathbb{V}\mathsf{ar}$. Essentially, such strings encode information that isol and $\overline{\mathsf{isol}}$ use to "keep track" of each other's nondeterministic or data-dependent choices. If they cannot do that, an isolated process and its coisolation run the risk of going "out of sync." To clarify this, suppose that we want to compose the $\{a\}$-isolation and $\{a\}$-coisolation of the process $r = a \cdot b + a \cdot c$ in parallel. For the sake of argument, suppose that isol and $\overline{\mathsf{isol}}$ take only a basic process as input and no string. We now demonstrate that this can go wrong. We have:

$$
\begin{array}{lll}
\mathsf{isol}(r) = & a \sqcup \xi(a) \cdot & \bar{\xi}(b) + a \sqcup \xi(a) \cdot \bar{\xi}(c) \\
\overline{\mathsf{isol}}(r) = \bar{\xi}(a) \cdot b \sqcup \xi(b) + & \bar{\xi}(a) \cdot c \sqcup \xi(c) &
\end{array}
$$

This means that $\mathsf{isol}(r)$ can erroneously synchronize its left-most multiaction $a \sqcup \xi(a)$ with the right-most multiaction $\bar{\xi}(a)$ of $\overline{\mathsf{isol}}(r)$, causing deadlock afterwards (because $\bar{\xi}(b)$ cannot synchronize with $\xi(c)$). To solve this problem, we use strings over $\{1\,,\,2\} \cup \mathbb{E}\mathsf{lem} \cup \mathbb{V}\mathsf{ar}$: essentially, we associate with every branch of the parse tree of a process a unique such string. This string encodes information about the structure of that process and its data bindings. Moreover, we ensure (e.g., by defining ξ and $\bar{\xi}$ as injective functions) that the isolation and the coisolation of a process synchronize auxiliary actions only if they belong to the same branch (in which case they have matching strings). For example:

$$
\begin{array}{lll}
\mathsf{isol}(\epsilon\,,\,r) = a \sqcup \xi_{11}(a) \cdot & \bar{\xi}_{12}(b) + a \sqcup \xi_{21}(a) \cdot & \bar{\xi}_{22}(c) \\
\overline{\mathsf{isol}}(\epsilon\,,\,r) = \bar{\xi}_{11}(a) \cdot b \sqcup \xi_{12}(b) + & \bar{\xi}_{21}(a) \cdot c \sqcup \xi_{22}(c) &
\end{array}
$$

[11] Strictly speaking, isol and $\overline{\mathsf{isol}}$ also take a substitution environment and a set of actions $\mathbb{A}$ as input.

$$
\begin{array}{lll}
\mathsf{isol}(w\,,\,a(\mathbf{d})) & = a(\mathbf{d}) \sqcup \xi_{w^\sharp}(a)(w^\flat) & \textbf{if } a \in \mathbb{A} \\
\mathsf{isol}(w\,,\,b(\mathbf{e})) & = \bar{\xi}_{w^\sharp}(b)(w^\flat) & \textbf{if } b \notin \mathbb{A} \\
\overline{\mathsf{isol}}(w\,,\,a(\mathbf{d})) & = \bar{\xi}_{w^\sharp}(a)(w^\flat) & \textbf{if } a \in \mathbb{A} \\
\overline{\mathsf{isol}}(w\,,\,b(\mathbf{e})) & = b(\mathbf{e}) \sqcup \xi_{w^\sharp}(b)(w^\flat) & \textbf{if } b \notin \mathbb{A} \\
\widehat{\mathsf{isol}}(w\,,\,\tau) & = \tau & \\
\widehat{\mathsf{isol}}(w\,,\,\beta \sqcup \gamma) & = \widehat{\mathsf{isol}}(w\,,\,\beta) \sqcup \widehat{\mathsf{isol}}(w\,,\,\gamma) & \\
\widehat{\mathsf{isol}}(w\,,\,\delta) & = \delta & \\
\widehat{\mathsf{isol}}(w\,,\,q + r) & = \widehat{\mathsf{isol}}(w1\,,\,q) + \widehat{\mathsf{isol}}(w2\,,\,r) & \\
\widehat{\mathsf{isol}}(w\,,\,q \cdot r) & = \widehat{\mathsf{isol}}(w1\,,\,q) \cdot \widehat{\mathsf{isol}}(w2\,,\,r) & \\
\widehat{\mathsf{isol}}(w\,,\,c \twoheadrightarrow q \diamond r) & = c \twoheadrightarrow \widehat{\mathsf{isol}}(w1\,,\,q) \diamond \widehat{\mathsf{isol}}(w2\,,\,r) & \\
\widehat{\mathsf{isol}}(w\,,\,\sum_{x\in D} q) & = \sum_{x\in D} \widehat{\mathsf{isol}}(wx\,,\,q) &
\end{array}
$$

Fig. 11. Definitions of isol and $\overline{\text{isol}}$. Let $\widehat{\text{isol}}$ range over the set $\{\text{isol}\,,\,\overline{\text{isol}}\}$.

$$
\begin{array}{ll}
1^\sharp & = 1 \\
2^\sharp & = 2 \\
d^\sharp\,,\,x^\sharp\,,\,\epsilon^\sharp & = \epsilon \\
(wv)^\sharp & = w^\sharp v^\sharp
\end{array}
$$

Fig. 12. Definition of $\sharp$.

$$
\begin{array}{ll}
1^\flat\,,\,2^\flat\,,\,\epsilon^\flat & = \epsilon \\
d^\flat & = d \\
x^\flat & = x \\
(wv)^\flat & = w^\flat v^\flat
\end{array}
$$

Fig. 13. Definition of $\flat$.

$$
\begin{array}{lll}
\mathsf{split}(w\,,\,p) & = ?(\mathsf{isol}(\epsilon\,,\,p) \parallel \overline{\mathsf{isol}}(\epsilon\,,\,p)) & \textbf{if } p \in \mathsf{Basic} \\
\mathsf{split}(w\,,\,P(\mathbf{d})) & = P^\dagger(\mathbf{d}) & \\
\mathsf{split}(w\,,\,q \oplus r) & = \mathsf{split}(w\,,\,q) \oplus \mathsf{split}(w\,,\,r) & \textbf{if } q \oplus r \notin \mathsf{Basic} \\
\mathsf{split}(w\,,\,c \twoheadrightarrow q \diamond r) & = c \twoheadrightarrow \mathsf{split}(w\,,\,q) \diamond \mathsf{split}(w\,,\,r) & \textbf{if } c \twoheadrightarrow q \diamond r \notin \mathsf{Basic} \\
\mathsf{split}(w\,,\,\sum_{x\in D} q) & = \sum_{x\in D} \mathsf{split}(w\,,\,q) & \textbf{if } \sum_{x\in D} q \notin \mathsf{Basic} \\
\mathsf{split}(w\,,\,f(q)) & = f(\mathsf{split}(w\,,\,q)) &
\end{array}
$$

Fig. 14. Definitions of split.

In this case, assuming some appropriate notion of synchronization that takes strings into account (we define this shortly), $a \sqcup \xi_{11}(a)$ can synchronize only with $\bar{\xi}_{11}(a)$ (they share the same string) and not with $\bar{\xi}_{21}(a)$ (different string). And so, these two processes do not go out of sync.

Let us now have a closer look at Fig. 11. Applied to a string w and a single action $a(\mathbf{d})$, depending on whether $\mathbb{A}$ contains a, isol and $\overline{\text{isol}}$ either compose or replace $a(\mathbf{d})$ with an auxiliary action using the substitution functions ξ and $\bar{\xi}$. However, because ξ and $\bar{\xi}$ have $\{1\,,\,2\}^* \times A$ as domain (see Definition 3), isol and $\overline{\text{isol}}$ cannot directly use w in ξ or $\bar{\xi}$: because $w \in (\{1\,,\,2\} \cup \mathbb{E}\mathsf{lem} \cup \mathbb{V}\mathsf{ar})^*$, isol and $\overline{\text{isol}}$ should first filter out the data elements and data variables possibly occurring in w. We introduce an operator denoted by $\sharp$ for that purpose. Fig. 12 shows its definition. Similarly, we introduce an operator denoted by $\flat$, which does the converse of $\sharp$: it filters symbols 1 and 2 from a string over $\{1\,,\,2\} \cup \mathbb{E}\mathsf{lem} \cup \mathbb{V}\mathsf{ar}$. Fig. 13 shows its definition. Functions isol and $\overline{\text{isol}}$ use $\flat$ to parameterize auxiliary actions with data. This parameterization ensures that the isolation and coisolation of a process of the form $\sum_{x\in D} q$ do not go out of sync (similar to what we saw in the example above). In Section 4.3, we exemplify this further.

Applied to a composite multiaction $\beta \sqcup \gamma$, isol and $\overline{\text{isol}}$ apply themselves recursively on β and γ without changing w. This differs for processes with a different main composition operator. For instance, for processes of the form $p + q$, isol and $\overline{\text{isol}}$ apply themselves recursively on $w1$ and $w2$ instead of w. This ensures that in their parallel composition, if appropriately synchronized, the process $\mathsf{isol}(w\,,\,p+q)$ can track which choice the process $\overline{\mathsf{isol}}(w\,,\,p+q)$ makes and vice versa as outlined above.

We make a final remark about the practical computability of isol and $\overline{\text{isol}}$. Strictly speaking, because we defined ξ and $\bar{\xi}$ as functions over $\{1\,,\,2\}^*$, those functions have infinite domains. This may seem problematic in practice, but fortunately, one can easily fix this. Start by observing that process terms consist of only finitely many operators and actions. This means that for $\mathsf{isol}(w\,,\,p)$ and $\overline{\mathsf{isol}}(w\,,\,p)$ to be defined (for some w and p), functions ξ and $\bar{\xi}$ must be defined for only finitely many strings (all of which have w as a prefix). One can compute this set of strings W in a preprocessing step that analyzes the syntax of p (essentially a dry run of isol or $\overline{\text{isol}}$). Then, before actually applying isol or $\overline{\text{isol}}$, define a finite substitution environment Ξ such that the domains of ξ and $\bar{\xi}$ contain only the strings in W. Thus, rather than one general substitution environment for all processes, we have a tailored substitution environment for every individual process (this generalizes straightforwardly to finite collections of processes). Henceforth, we always assume a finite yet sufficient substitution environment Ξ when we apply isol or $\overline{\text{isol}}$ to a (collection of) process(es).

Splitting We build the definition of function split—the actual splitting procedure, so to speak—on top of functions isol and $\overline{\text{isol}}$. Fig. 14 shows its definition.

We also introduce an auxiliary operator, denoted by ?, which represents and ensures "appropriate synchronization" among auxiliary actions: it takes care of the communication, hiding, and blocking necessary to synchronize auxiliary actions

such that split preserves semantics (as exemplified in Section 4.1). Recall that we implicitly quantify universally over all substitution environments Ξ in definitions to avoid heavy notation. Then[12]:

Definition 4. $?(p) = \partial_{\mathsf{img}(\Xi)}(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\mathsf{comm}(\Xi)}(p)))$.

Example. Suppose $\Xi = \langle A\,,\,\mathtt{tau}\,,\,\xi\,,\,\bar{\xi}\rangle$ is the substitution environment defined in the Example on page 55 and $\mathbb{A} = \{\mathtt{foo}\}$. Then:

- From the Example on page 56:
  - $\mathsf{img}(\Xi) = \bigcup_{w\in\{1\,,\,2\}^*}\{\mathtt{x\_foo\_}w\,,\,\overline{\mathtt{x\_foo\_}w}\}$
  - $\mathsf{comm}(\Xi) = \{\mathtt{x\_foo\_}w \sqcup \overline{\mathtt{x\_foo\_}w} \to \mathtt{tau} \mid w \in \{1\,,\,2\}^*\}$
- $\begin{aligned}[t]
\mathcal{T}_{\{\mathtt{foo}\}}(\mathsf{split}(\mathtt{foo})) &= \mathcal{T}_{\{\mathtt{foo}\}}(\mathsf{split}(\mathtt{foo})) \\
&= \mathcal{T}_{\{\mathtt{foo}\}}(?(\mathsf{isol}(\epsilon\,,\,\mathtt{foo}) \parallel \overline{\mathsf{isol}}(\epsilon\,,\,\mathtt{foo}))) \\
&= \mathcal{T}_{\{\mathtt{foo}\}}(?(\mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \parallel \overline{\mathtt{x_foo}_\epsilon})) \\
&= \mathcal{T}_{\{\mathtt{foo}\}}(\partial_{\bigcup_{w\in\{1\,,\,2\}^*}\{\mathtt{x_foo_}w\,,\,\overline{\mathtt{x_foo_}w}\}}(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\{\mathtt{x_foo_}w\sqcup\overline{\mathtt{x_foo_}w}\to\mathtt{tau}\mid w\in\{1\,,\,2\}^*\}}(\\
&\qquad \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \parallel \overline{\mathtt{x_foo}_\epsilon})))) \\
&\simeq \mathcal{T}_{\{\mathtt{foo}\}}(\partial_{\bigcup_{w\in\{1\,,\,2\}^*}\{\mathtt{x_foo_}w\,,\,\overline{\mathtt{x_foo_}w}\}}(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\{\mathtt{x_foo_}w\sqcup\overline{\mathtt{x_foo_}w}\to\mathtt{tau}\mid w\in\{1\,,\,2\}^*\}}(\\
&\qquad \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \between \overline{\mathtt{x_foo}_\epsilon} + \\
&\qquad \overline{\mathtt{x_foo}_\epsilon} \between \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon + \\
&\qquad \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \mid \overline{\mathtt{x_foo}_\epsilon})))) \\
&\simeq \mathcal{T}_{\{\mathtt{foo}\}}(\partial_{\bigcup_{w\in\{1\,,\,2\}^*}\{\mathtt{x_foo_}w\,,\,\overline{\mathtt{x_foo_}w}\}}(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\{\mathtt{x_foo_}w\sqcup\overline{\mathtt{x_foo_}w}\to\mathtt{tau}\mid w\in\{1\,,\,2\}^*\}}(\\
&\qquad \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \cdot \overline{\mathtt{x_foo}_\epsilon} + \overline{\mathtt{x_foo}_\epsilon} \cdot \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon + \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \sqcup \overline{\mathtt{x_foo}_\epsilon})))) \\
&\simeq \mathcal{T}_{\{\mathtt{foo}\}}(\partial_{\bigcup_{w\in\{1\,,\,2\}^*}\{\mathtt{x_foo_}w\,,\,\overline{\mathtt{x_foo_}w}\}}(\mathcal{T}_{\{\mathtt{tau}\}}(\\
&\qquad \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \cdot \overline{\mathtt{x_foo}_\epsilon} + \overline{\mathtt{x_foo}_\epsilon} \cdot \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon + \mathtt{foo} \sqcup \mathtt{tau})))) \\
&\simeq \mathcal{T}_{\{\mathtt{foo}\}}(\partial_{\bigcup_{w\in\{1\,,\,2\}^*}\{\mathtt{x_foo_}w\,,\,\overline{\mathtt{x_foo_}w}\}}(\\
&\qquad \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon \cdot \overline{\mathtt{x_foo}_\epsilon} + \overline{\mathtt{x_foo}_\epsilon} \cdot \mathtt{foo} \sqcup \mathtt{x_foo}_\epsilon + \mathtt{foo} \sqcup \tau)) \\
&\simeq \mathcal{T}_{\{\mathtt{foo}\}}(\delta + \delta + \mathtt{foo} \sqcup \tau) \\
&\simeq \mathcal{T}_{\{\mathtt{foo}\}}(\mathtt{foo})
\end{aligned}$

(We ignore data in this simple example.) □

The definition of $\mathsf{split}(w\,,\,p)$ for $p = P$ may seem odd and requires more explanation, because we make a number of tacit assumptions. First, we assume that if a process reference R occurs in some process q, there exists also a process definition $R = r$ (otherwise, q has no meaning). Second, we adopt the notational convention that every process reference with a superscript † refers to a process definition with a body to which we applied split (for the empty string). For example, $R^\dagger = \mathsf{split}(\epsilon\,,\,r)$. Now, the definition of $\mathsf{split}(w\,,\,P)$ makes more sense: it means that we replace process references in a split process with process references that refer to other split processes. In that way, the application of split propagates through process definitions. In Section 5.4, we prove the correctness of this definition.

4.3. More examples

To illustrate the usage of split, we give three more examples in this subsection. For the sake of clarity, we use concrete action names for both original actions and auxiliary actions as follows. Define:

$$\mathbb{A}\mathsf{ct} = \underbrace{\{\mathtt{foo}\,,\,\mathtt{bar}\,,\,\mathtt{baz}\}}_{\text{original actions}} \cup \underbrace{\left\{\begin{matrix}\mathtt{x\_foo\_1}\,,\,\mathtt{x\_bar\_2}\,,\,\mathtt{x\_bar\_21}\,,\,\mathtt{x\_foo\_22}\,,\,\mathtt{x\_baz} \\ \overline{\mathtt{x\_foo\_1}}\,,\,\overline{\mathtt{x\_bar\_2}}\,,\,\overline{\mathtt{x\_bar\_21}}\,,\,\overline{\mathtt{x\_foo\_22}}\,,\,\overline{\mathtt{x\_baz}}\end{matrix}\right\}}_{\text{auxiliary actions}}$$

[12] In (detailed versions of) proofs of propositions, lemmas, and theorems in Section 5, we use the following axioms for ? (except Q1, which we give for the sake of a comprehensive presentation of the properties of ?):

Q1	$?(\tau) \simeq \tau$	Q3	$?(q + r) \simeq ?(q) + ?(r)$	Q5	$?(\sum_{x\in D} q) \simeq \sum_{x\in D} ?(q)$
Q2	$?(\delta) \simeq \delta$	Q4	$?(q \cdot r) \simeq ?(q) \cdot ?(r)$		

The validity of these axioms follows directly from axioms in Fig. 6: Axiom Q1 follows from C1, H1, and B1; Q2 follows from F1; Q3 follows from F2; Q4 follows from F3; and Q5 follows from F4.

Furthermore, let $\mathbb{A} = \{\texttt{foo}, \texttt{baz}\}$ (i.e., we split along `foo` and `baz`), define ξ as follows:

$$\xi_1(\texttt{foo}) = \texttt{x\_foo\_1} \qquad \xi_2(\texttt{bar}) = \texttt{x\_bar\_2} \qquad \xi_{21}(\texttt{bar}) = \texttt{x\_bar\_21}$$
$$\xi_{22}(\texttt{foo}) = \texttt{x\_foo\_22} \qquad \xi_\epsilon(\texttt{baz}) = \texttt{x\_baz}$$

and define $\bar{\xi}$ analogously.

Example 1 Let $p_1 = \texttt{foo}(1, 2) + \texttt{bar}(3)$. We derive $\mathsf{split}(\epsilon, p_1)$ as follows.

$$\begin{aligned}
& \mathsf{split}(\epsilon, p_1) \\
= \;& \mathsf{split}(\epsilon, \texttt{foo}(1, 2) + \texttt{bar}(3)) \\
= \;& ?(\mathsf{isol}(\epsilon, \texttt{foo}(1, 2) + \texttt{bar}(3)) \parallel \overline{\mathsf{isol}}(\epsilon, \texttt{foo}(1, 2) + \texttt{bar}(3))) \\
= \;& ?(\mathsf{isol}(1, \texttt{foo}(1, 2)) + \mathsf{isol}(2, \texttt{bar}(3)) \parallel \overline{\mathsf{isol}}(1, \texttt{foo}(1, 2)) + \overline{\mathsf{isol}}(2, \texttt{bar}(3))) \\
= \;& ?((\texttt{foo}(1, 2) \sqcup \xi_1(\texttt{foo}) + \bar{\xi}_2(\texttt{bar})) \parallel (\bar{\xi}_1(\texttt{foo}) + \texttt{bar}(3) \sqcup \bar{\xi}_2(\texttt{bar}))) \\
= \;& ?((\texttt{foo}(1, 2) \sqcup \texttt{x_foo_1} + \texttt{x_bar_2}) \parallel (\overline{\texttt{x_foo_1}} + \texttt{bar}(3) \sqcup \overline{\texttt{x_bar_2}})) \\
= \;& ?((\texttt{foo}(1, 2) \sqcup \texttt{x_foo_1} + \quad \texttt{x_bar_2}) \parallel \\
& \quad (\quad \overline{\texttt{x_foo_1}} + \texttt{bar}(3) \sqcup \overline{\texttt{x_bar_2}}) \quad)
\end{aligned}$$

Note that the auxiliary actions $\texttt{x\_foo\_1}$, $\overline{\texttt{x\_foo\_1}}$, $\texttt{x\_bar\_2}$, and $\overline{\texttt{x\_bar\_2}}$ have no data parameters in this example, because none of the strings to which we apply ξ and $\bar{\xi}$ contain symbols outside $\{1, 2\}$. (In those case, by the definition of $\flat$, auxiliary actions have no parameters.) Next, we consider an example in which data do play a role.

Example 2 Let

$$p_2 = \sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \texttt{foo}(\mathit{true}) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false})$$

for $D_1 = \{i \mid 6 \leqslant i \leqslant 496\}$ and $D_2 = \{1, 2\}$. We derive $\mathsf{split}(\epsilon, p_2)$ as follows.

$$\begin{aligned}
& \mathsf{split}(\epsilon, p_2) \\
= \;& \mathsf{split}(\epsilon, \textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \texttt{foo}(\mathit{true}) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false})) \\
= \;& ?(\mathsf{isol}(\epsilon, \textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \texttt{foo}(\mathit{true}) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false})) \parallel \\
& \quad \overline{\mathsf{isol}}(\epsilon, \textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \texttt{foo}(\mathit{true}) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false}))) \\
= \;& ?(\textstyle\sum_{x \in D_1} \mathsf{isol}(x, x \leqslant 28 \rightarrowtriangle \texttt{foo}(\mathit{true}) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false})) \parallel \\
& \quad \textstyle\sum_{x \in D_1} \overline{\mathsf{isol}}(x, x \leqslant 28 \rightarrowtriangle \texttt{foo}(\mathit{true}) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false}))) \\
= \;& ?(\textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \mathsf{isol}(x1, \texttt{foo}(\mathit{true})) \diamond \mathsf{isol}(x2, \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false})) \parallel \\
& \quad \textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \overline{\mathsf{isol}}(x1, \texttt{foo}(\mathit{true})) \diamond \overline{\mathsf{isol}}(x2, \sum_{y \in D_2} \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false}))) \\
= \;& ?(\textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \mathsf{isol}(x1, \texttt{foo}(\mathit{true})) \diamond \sum_{y \in D_2} \mathsf{isol}(x2y, \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false})) \parallel \\
& \quad \textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \overline{\mathsf{isol}}(x1, \texttt{foo}(\mathit{true})) \diamond \sum_{y \in D_2} \overline{\mathsf{isol}}(x2y, \texttt{bar}(x, y) \cdot \texttt{foo}(\mathit{false}))) \\
= \;& ?(\textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \mathsf{isol}(x1, \texttt{foo}(\mathit{true})) \diamond \sum_{y \in D_2} \mathsf{isol}(x2y1, \texttt{bar}(x, y)) \cdot \mathsf{isol}(x2y2, \texttt{foo}(\mathit{false})) \parallel \\
& \quad \textstyle\sum_{x \in D_1} x \leqslant 28 \rightarrowtriangle \overline{\mathsf{isol}}(x1, \texttt{foo}(\mathit{true})) \diamond \sum_{y \in D_2} \overline{\mathsf{isol}}(x2y1, \texttt{bar}(x, y) \cdot \overline{\mathsf{isol}}(x2y2, \texttt{foo}(\mathit{false})))) \\
= \;& ?(\textstyle\sum_{x \in D_1} x \leqslant 28 \\
& \quad \rightarrowtriangle \texttt{foo}(\mathit{true}) \sqcup \texttt{x_foo_1}(x) \diamond \sum_{y \in D_2} \qquad \overline{\texttt{x_bar_21}}(x, y) \cdot \texttt{foo}(\mathit{false}) \sqcup \texttt{x_foo_22}(x, y) \parallel \\
& \quad \textstyle\sum_{x \in D_1} x \leqslant 28 \\
& \quad \rightarrowtriangle \qquad \overline{\texttt{x_foo_1}}(x) \diamond \sum_{y \in D_2} \texttt{bar}(x, y) \sqcup \texttt{x_bar_21}(x, y) \cdot \qquad \overline{\texttt{x_foo_22}}(x, y))
\end{aligned}$$

This example demonstrates how the splitting procedure handles data-dependent processes. Furthermore, based on this example, we can illustrate an important property guaranteed by the data parameters of auxiliary actions: the isolation of p_2 (left/above of $\parallel$) and the coisolation of p_2 (right/below of $\parallel$) terminate successfully only if they bind x (and later y) to the same value. To see this, suppose that the isolation binds x to 4, while the coisolation binds x to 28 (such that these processes take the same branch of the conditional choice). Then, because the communication operator Γ embedded in ? requires that communicating actions have the same data parameters (see Section 2.3), $\texttt{x\_foo\_1}(4)$ and $\texttt{x\_foo\_1}(28)$ cannot synchronize with each other. This in turn causes deadlock (effected by the blocking operator in ?). In contrast, if both the isolation and the coisolation bind x to 4, the auxiliary actions parameterized by x can synchronize, after which the whole process terminates successfully.

Example 3 Let $p_3 = p_1 \parallel \mathtt{baz} \cdot P_4$ for process definitions $P_4 = \mathtt{baz} \cdot P_4$ and $P_4^\dagger = \mathsf{split}(\epsilon\,,\ \mathtt{baz} \cdot P_4)$. We derive $\mathsf{split}(\epsilon\,,\ p_3)$ as follows.

$$
\begin{aligned}
& \mathsf{split}(\epsilon\,,\ p_3) \\
= \ & \mathsf{split}(\epsilon\,,\ p_1 \parallel \mathtt{baz} \cdot P_4) \\
= \ & \mathsf{split}(\epsilon\,,\ p_1) \parallel \mathsf{split}(\epsilon\,,\ \mathtt{baz} \cdot P_4) \\
= \ & \mathsf{split}(\epsilon\,,\ p_1) \parallel (\mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot \mathsf{split}(\epsilon\,,\ P_4)) \\
= \ & \mathsf{split}(\epsilon\,,\ p_1) \parallel (\mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\dagger) \\
= \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}} \quad) \parallel (\mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\dagger)
\end{aligned}
$$

To further rewrite this process, we introduce the following process definitions: $P_4^\ddagger = ?(P_4^\S \parallel \overline{P}_4^\S)$ and $P_4^\S = \mathsf{isol}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\S$ and $\overline{P}_4^\S = \overline{\mathsf{isol}}(\epsilon\,,\ \mathtt{baz}) \cdot \overline{P}_4^\S$. Using RSP (see Section 2.3), one can show that $P_4^\dagger$ is provably equal to $P_4^\ddagger$,[13] which enables the following rewrites:

$$
\begin{aligned}
\simeq \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}} \quad) \parallel (\mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\ddagger) \\
\simeq \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}} \quad) \parallel (\mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot ?(P_4^\S \parallel \overline{P}_4^\S))
\end{aligned}
$$

Using [28, Appendix D, Lemma 6, page 87], and by afterward expanding the definitions of isol, $\overline{\mathsf{isol}}$, and the substitution functions, we further rewrite as follows:

$$
\begin{aligned}
\simeq \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}} \quad) \parallel ?((\mathsf{isol}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\S) \parallel (\overline{\mathsf{isol}}(\epsilon\,,\ \mathtt{baz}) \cdot \overline{P}_4^\S)) \\
= \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}} \quad) \parallel ?((\mathtt{baz} \sqcup \xi_\epsilon(\mathtt{baz}) \cdot P_4^\S) \parallel (\overline{\xi}_\epsilon(\mathtt{baz}) \cdot \overline{P}_4^\S)) \\
= \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}} \quad) \parallel ?((\mathtt{baz} \sqcup \mathtt{x\_baz} \cdot P_4^\S) \parallel (\overline{\mathtt{x\_baz}} \cdot \overline{P}_4^\S))
\end{aligned}
$$

This process has the less attractive feature that ? does not occur as a top-level operator. Intuitively, this means that we have to ensure appropriate synchronization at multiple places. Fortunately, using mCRL2's *alphabet axioms* [22, Section 5.6], we can rewrite this process as follows:

$$
\begin{aligned}
\simeq \ & ?((\mathtt{foo}(1\,,\ 2) \sqcup \mathtt{x\_foo\_1} + \quad\quad \mathtt{x\_bar\_2}) \parallel \\
& \quad (\quad\quad \overline{\mathtt{x\_foo\_1}} + \mathtt{bar}(3) \sqcup \overline{\mathtt{x\_bar\_2}}) \parallel \\
& \quad (\quad\quad\quad\quad\quad\quad \mathtt{baz} \sqcup \mathtt{x\_baz} \cdot P_4^\S) \parallel \\
& \quad (\quad\quad\quad\quad\quad\quad\quad\quad \overline{\mathtt{x\_baz}} \cdot \overline{P}_4^\S) \quad)
\end{aligned}
$$

Basically, the alphabet axioms state under what conditions unary operators such as Γ, $\mathcal{T}$, and ∂ are preserved by $\parallel$, under what conditions they commute with each other, and under what conditions one can safely omit or add them without changing the semantics of a process. Exactly those properties allow us to push ?, which consists of Γ, $\mathcal{T}$, and ∂, outward to the top-level.[14]

[13] First, one should show that $P_4^\ddagger$ is provably equal to $\mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\ddagger$ as follows:

$$P_4^\ddagger \simeq ?(P_4^\S \parallel \overline{P}_4^\S) \simeq ?((\mathsf{isol}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\S) \parallel (\overline{\mathsf{isol}}(\epsilon\,,\ \mathtt{baz}) \cdot \overline{P}_4^\S)) \simeq \mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot ?(P_4^\S \parallel \overline{P}_4^\S) \simeq \mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot P_4^\ddagger$$

(To prove the third step, use [28, Appendix D, Lemma 6, page 87].) Now, one can straightforwardly use RSP with the following guarded process operator: $\Phi(X) = \mathsf{split}(\epsilon\,,\ \mathtt{baz}) \cdot X$. Generally, this property holds not only for single actions (such as $\mathtt{baz}$) but for every basic process.

[14] This works as follows. We have a process of the form $?(p) \parallel ?(q)$. By expanding ?, we get $\partial_B \mathsf{img}(\Xi)(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\mathsf{comm}(\Xi)}(p))) \parallel \partial_{\mathsf{img}(\Xi)}(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\mathsf{comm}(\Xi)}(q)))$. Using Axioms DL2 and TL2 in [22, Section 5.6], which state that ∂ and $\mathcal{T}$ are preserved by $\parallel$ (i.e., ∂ and $\mathcal{T}$ are homomorphic with respect to $\parallel$), we can rewrite this to $\partial_B \mathsf{img}(\Xi)(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\mathsf{comm}(\Xi)}(p) \parallel \Gamma_{\mathsf{comm}(\Xi)}(q)))$. To show that also Γ is preserved by $\parallel$ *in this case* (not generally!), we first observe that $\mathsf{Act}(p) \cap \mathsf{Act}(q) = \emptyset$. Consequently, the auxiliary actions occurring in p do not occur in q and vice versa. Moreover, if an auxiliary action does occur in p (resp. q), also its dual occurs in p (resp. q). By recalling that every communication rule in $\mathsf{comm}(\Xi)$ has a pair of an auxiliary action and its dual as left-hand side (see Fig. 10), we can now split $\mathsf{comm}(\Xi)$ into three disjoint sets C_p, C_q, $C_\emptyset$ such that: the communications rules in C_p contain auxiliary actions occurring only in p, the communication rules in C_q contain auxiliary actions occurring only in q, and the communication rules in $C_\emptyset = \mathsf{comm}(\Xi) \setminus (C_p \cup C_q)$ contain

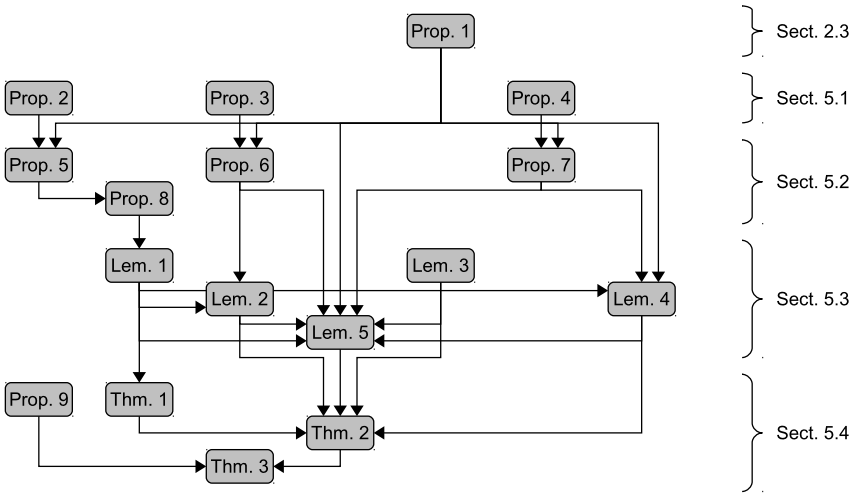


Fig. 15. Dependency graph of propositions, lemmas, and correctness theorems.

5. Properties of the splitting procedure

In this section, we prove the correctness of the definitions presented in Section 4: we establish that processes p and $\mathsf{split}(p)$ are provably equal from the axioms of mCRL2 (see Section 2.3). This implies that p and $\mathsf{split}(p)$ behave indistinguishably under any behavioral congruence satisfying those axioms (e.g., strong bisimilarity). Fig. 15 shows a graph of how the propositions, lemmas, and theorems presented in this section depend on each other. First, the propositions in Section 5.1 state properties about when split multiactions cause deadlock. Second, the propositions in Section 5.2 generalize those previous propositions from multiactions to basic processes. Third, in Section 5.3, we present a lemma (Lemma 1) that states that (co)isolated processes execute in lockstep when appropriately synchronized by ? and four lemmas that state preservation properties of split. Finally, we use the lemmas in Section 5.3 in Section 5.4 to prove our correctness theorems for multiactions (Theorem 1), basic processes (Theorem 2), and arbitrary process specifications (Theorem 3).

Notation In all propositions, we implicitly quantify universally over all elements over which symbols occurring in those propositions range, unless stated otherwise. The same applies to lemmas and theorems. We use square brackets as meaningless delimiters.

Although each of the premises and consequents in the propositions in this section serves a purpose, they sometimes make these propositions heavy on notation and difficult to parse. Therefore, to highlight the essence of a proposition, we sometimes gray out those parts that seem less important for conveying the key result. (The parts are essential for proving the result in detail, though.) The same applies to lemmas and theorems.

---

auxiliary actions occurring neither in p nor in q. Using Axioms CL2, CL3, and CL4 in [22, Section 5.6], we can rewrite $\Gamma_{\mathsf{comm}(\Xi)}(p) \parallel \Gamma_{\mathsf{comm}(\Xi)}(q)$ as follows (we implicitly apply commutativity of $\parallel$):

$$
\begin{aligned}
& \Gamma_{\mathsf{comm}(\Xi)}(p) \parallel \Gamma_{\mathsf{comm}(\Xi)}(q) \\
= \;& \Gamma_{C_\emptyset \cup C_p \cup C_q}(p) \parallel \Gamma_{C_\emptyset \cup C_p \cup C_q}(q) \\
\overset{\text{CL2}}{\simeq} \;& \Gamma_{C_\emptyset}\Gamma_{C_p}\Gamma_{C_q}(p) \parallel \Gamma_{C_\emptyset}\Gamma_{C_p}\Gamma_{C_q}(q) \\
\overset{\text{CL3}}{\simeq} \;& \Gamma_{C_\emptyset}(\Gamma_{C_\emptyset}\Gamma_{C_p}\Gamma_{C_q}(p) \parallel \Gamma_{C_p}\Gamma_{C_q}(q)) \\
\overset{\text{CL4}}{\simeq} \;& \Gamma_{C_\emptyset}(\Gamma_{C_p}\Gamma_{C_q}(p) \parallel \Gamma_{C_p}\Gamma_{C_q}(q)) \\
\overset{\text{CL3}}{\simeq} \;& \Gamma_{C_\emptyset}\Gamma_{C_p}(\Gamma C C_p \Gamma_{C_q}(p) \parallel \Gamma_{C_q}(q)) \\
\overset{\text{CL4}}{\simeq} \;& \Gamma_{C_\emptyset}\Gamma_{C_p}(\Gamma_{C_q}(p) \parallel \Gamma_{C_q}(q)) \\
\overset{\text{CL3}}{\simeq} \;& \Gamma_{C_\emptyset}\Gamma_{C_p}\Gamma_{C_q}(p \parallel \Gamma_{C_q}(q)) \\
\overset{\text{CL4}}{\simeq} \;& \Gamma_{C_\emptyset}\Gamma_{C_p}\Gamma_{C_q}(p \parallel q) \\
\overset{\text{CL2}}{\simeq} \;& \Gamma_{C_\emptyset \cup C_p \cup C_q}(p \parallel q) \\
= \;& \Gamma_{\mathsf{comm}(\Xi)}(p \parallel q)
\end{aligned}
$$

Applying this to our previous result, we get $\partial_{\mathsf{img}(\Xi)}(\mathcal{T}_{\{\mathtt{tau}\}}(\Gamma_{\mathsf{comm}(\Xi)}(p \parallel q)))$. Hence, we conclude $?(p) \parallel ?(q) \simeq ?(p \parallel q)$.

$$
\begin{array}{rl}
a(\mathbf{d})\,,\ \delta \in \mathsf{TauFree} & \\
\beta \sqcup \gamma \in \mathsf{TauFree} & \textbf{iff } \left[\beta \in \mathsf{TauFree} \textbf{ and } \gamma \in \mathsf{TauFree}\right] \\
q \oplus r\,,\ c \twoheadrightarrow q \diamond r \in \mathsf{TauFree} & \textbf{iff } \left[p \in \mathsf{TauFree} \textbf{ and } q \in \mathsf{TauFree}\right] \\
\sum_{x \in D} q\,,\ f(q) \in \mathsf{TauFree} & \textbf{iff } q \in \mathsf{TauFree}
\end{array}
$$

Fig. 16. Definition of TauFree.

5.1. Simple properties I: deadlock caused by split multiactions

In this subsection, we formulate three propositions that state properties about when split multiactions cause deadlock. Essentially, these propositions formalize when the "appropriate synchronization" operator ? blocks auxiliary actions whose individual execution "makes no sense" (see Section 4.1).

Proposition 2 states that every appropriately synchronized lone (co)isolated multiaction $?(\widehat{\mathsf{isol}}(w\,,\,\alpha))$ causes deadlock. In the formulation of the premise, we write $\alpha \in \mathsf{TauFree}$ (defined in Fig. 16) to express that τ does not occur syntactically in α. Variants of this requirement appear in (nearly) all subsequent propositions, lemmas, and theorems. Fortunately, they limit the applicability of our results only marginally, because τ usually does not occur syntactically in processes (but instead results from hiding). The premise of Proposition 2 also ensures that the domain of the substitution environment contains the actions in α; otherwise, $\widehat{\mathsf{isol}}(w\,,\,\alpha)$ has no meaning.

Proposition 2 *($\widehat{\mathsf{isol}}$-multiactions cause deadlock).*

$$\left[\alpha \in \mathsf{TauFree} \textbf{ and } \mathsf{Act}(\alpha) \subseteq \mathsf{dom}(\Xi)\right] \textbf{ implies } ?(\widehat{\mathsf{isol}}(w\,,\,\alpha)) \simeq \delta$$

To understand why this proposition holds for $\widehat{\mathsf{isol}} = \mathsf{isol}$ (the $\widehat{\overline{\mathsf{isol}}}$ case works similar), observe that every isolated multiaction contains at least one auxiliary action $\widehat{\xi}$ (this follows immediately from the definition of isol). Now, reasoning toward a contradiction, suppose that also the dual of $\widehat{\xi}$ occurs in $?(\mathsf{isol}(w\,,\,\alpha))$. Then, the content of $\mathbb{A}$ must have changed between the construction of $\widehat{\xi}$ and its dual or vice versa (otherwise, isol produces either always $\widehat{\xi}$ or always its dual). But the content of $\mathbb{A}$ remains constant across applications of isol, so $\mathbb{A}$ cannot have changed. Hence, $\widehat{\xi}$ has no dual in $?(\mathsf{isol}(w\,,\,\alpha))$. This means that $\Gamma_{\mathsf{comm}(\Xi)}$ in ? does not affect $\widehat{\xi}$ (because the communications in $\mathsf{comm}(\Xi)$ involve only pairs of an auxiliary action and its dual). Also $\mathcal{T}_{\{\mathtt{tau}\}}$ in ? does not affect $\widehat{\xi}$ (because auxiliary actions differ from `tau` by Definition 3). This leaves us with $\partial_{\mathsf{img}(\Xi)}$, which does affect $\widehat{\xi}$: it blocks it. The resulting deadlock then propagates through the entire multiaction. See [28, Appendix B, page 58], for a detailed proof.

Proposition 3 states that the synchronous composition of an isolated multiaction and a coisolated multiaction *under different strings over* $\{1\,,\,2\}$ causes deadlock.

Proposition 3 *(Composed* isol- *and* $\overline{\mathsf{isol}}$*-multiactions cause deadlock, I).*

$$\left[\begin{array}{c}\beta\,,\ \gamma \in \mathsf{TauFree} \textbf{ and} \\ \mathsf{Act}(\beta)\,,\ \mathsf{Act}(\gamma) \subseteq \mathsf{dom}(\Xi) \\ \textbf{and } v^\sharp \neq u^\sharp\end{array}\right] \textbf{ implies } ?(\mathsf{isol}(v\,,\,\beta) \mid \overline{\mathsf{isol}}(u\,,\,\gamma)) \simeq \delta$$

The validity of this proposition crucially depends on the injectivity of substitution functions (see Definition 3). Essentially, this injectivity ensures that the auxiliary actions in $\mathsf{isol}(v\,,\,\beta)$ and $\overline{\mathsf{isol}}(u\,,\,\gamma)$ come from different pools: $\mathsf{isol}(v\,,\,\beta)$ and $\overline{\mathsf{isol}}(u\,,\,\gamma)$ have neither auxiliary actions nor their duals in common. Moreover, by similar reasoning as for Proposition 2, we can establish that $\overline{\mathsf{isol}}(u\,,\,\gamma)$ contains an auxiliary action $\widehat{\xi}$ but not its dual (the same holds for $\mathsf{isol}(v\,,\,\beta)$ but we do not need it). Thus, neither $\mathsf{isol}(v\,,\,\beta)$ nor $\overline{\mathsf{isol}}(u\,,\,\gamma)$ contains the dual of $\widehat{\xi}$. Then, again by similar reasoning as for Proposition 2, we can establish that $\Gamma_{\mathsf{comm}(\Xi)}$ and $\mathcal{T}_{\{\mathtt{tau}\}}$ in ? do not affect $\widehat{\xi}$ while $\partial_{\mathsf{img}(\Xi)}$ does. See [28, Appendix B, page 59], for a detailed proof.

Proposition 4 states that the synchronous composition of an isolated multiaction and a coisolated multiaction *under different data* causes deadlock.

Proposition 4 *(Composed* isol- *and* $\overline{\mathsf{isol}}$*-multiactions cause deadlock, II).*

$$\left[\begin{array}{c}\beta\,,\ \gamma \in \mathsf{TauFree} \textbf{ and} \\ \mathsf{Act}(\beta)\,,\ \mathsf{Act}(\gamma) \subseteq \mathsf{dom}(\Xi) \\ \textbf{and } e \neq f\end{array}\right] \textbf{ implies } ?(\mathsf{isol}(wev\,,\,\beta) \mid \overline{\mathsf{isol}}(wfu\,,\,\gamma)) \simeq \delta$$

Although similar to Proposition 3, we prove the validity of this proposition rather differently. In Proposition 3 (and also in Proposition 2), deadlock occurred due to lone auxiliary actions. But in this case, it can happen that all auxiliary actions occur with their dual (e.g., if $\beta = \gamma$ and $v = u$). Thus, we need a different strategy. To that end, observe that the premise of Proposition 4 ensures that the data parameters of an auxiliary action and its dual differ (because $e \neq f$). For instance, if $b \in \mathbb{A}$, we have $\mathsf{isol}(e\,,\,b) = b \sqcup \xi_\epsilon(b)(e)$ and $\overline{\mathsf{isol}}(f\,,\,b) = \overline{\xi}_\epsilon(b)(f)$. Now, even though $\xi_\epsilon(b)(e)$ and $\overline{\xi}_\epsilon(b)(f)$ are duals,

$\Gamma_{\mathsf{comm}(\Xi)}$ in ? does not affect their composition, because e and f differ (see Axiom C1 and the definition of $\mathcal{C}$ in Section 2.3). Because also $\mathcal{T}_{\{\mathtt{tau}\}}$ in ? does not affect these auxiliary actions by the same reasoning as before, again, we end up with $\partial_{\mathsf{img}(\Xi)}$, which blocks $\xi_\epsilon(b)(e)$ and $\overline{\xi}_\epsilon(b)(f)$. We can generalize this argument to arbitrary multiactions. See [28, Appendix B, page 62], for a detailed proof.

5.2. Simple properties II: deadlock caused by split basic processes

Next, we generalize the propositions in the previous subsection from multiactions to basic processes. Each of the proofs of these generalizations exploits the observation that for every (co)isolated basic process $\widehat{\mathsf{isol}}(w\,,\,p)$, there exists a provably equal process with the following structure: $\sum_{i=1}^{n} \widehat{\mathsf{isol}}(w w_i\,,\,\boldsymbol{\alpha}_i) + \sum_{i=1}^{n'} (\widehat{\mathsf{isol}}(w w'_i\,,\,\boldsymbol{\alpha}'_i) \cdot p'_i)$. Essentially, to establish that such processes cause deadlock, it suffices to show that $\widehat{\mathsf{isol}}(w w_i\,,\,\boldsymbol{\alpha}_i)$ and $\widehat{\mathsf{isol}}(w w'_i\,,\,\boldsymbol{\alpha}'_i)$ cause deadlock for all relevant i (because of Axioms A6 and A7 in Fig. 6). One can show this by applying (some of) the propositions from Section 5.1 for each such i.

The premise of each of the following propositions contains a variant of the requirement $\mathsf{Bound}(p) \pitchfork w = \emptyset$. The $\pitchfork$ operator denotes the intersection between the elements in a set (e.g., $\mathsf{Bound}(p)$) and the individual symbols of a string (e.g., w).[15] Thus, $\mathsf{Bound}(p) \pitchfork w = \emptyset$ means that the data variables that will become bound in p may not intersect with any of the data variables occurring in w. We forbid this, because if a data variable x occurs in w, this intuitively means that x already has been bound (due to how isol and $\overline{\mathsf{isol}}$ build strings). In other words, if $\mathsf{Bound}(p)$ and the elements in w intersect, p rebinds a data variable, which it should not. The requirement $\mathsf{Bound}(p) \pitchfork w = \emptyset$ has little consequences in practice: typically, $w = \epsilon$, in which case it holds vacuously. (Moreover, if necessary, one can avoid rebinding with an α-conversion preprocessing step.)

Proposition 5 *($\widehat{\mathsf{isol}}$-processes cause deadlock).*

$$\begin{bmatrix} p \in \mathsf{Basic}\ \textbf{and}\ p \in \mathsf{TauFree}\ \textbf{and} \\ \mathsf{Act}(p) \subseteq \mathsf{dom}(\Xi)\ \textbf{and}\ \mathsf{Bound}(p) \pitchfork w = \emptyset \end{bmatrix}\ \textbf{implies}\ \ ?(\widehat{\mathsf{isol}}(w\,,\,p)) \simeq \delta$$

See [28, Appendix C, page 76], for a detailed proof.

Proposition 6 *(Composed* isol- *and* $\overline{\mathsf{isol}}$*-processes cause deadlock, I).*

$$\begin{bmatrix} q\,,\,r \in \mathsf{Basic}\ \textbf{and}\ q\,,\,r \in \mathsf{TauFree}\ \textbf{and}\ \mathsf{Act}(q)\,,\,\mathsf{Act}(r) \subseteq \mathsf{dom}(\Xi) \\ \textbf{and}\ \mathsf{Bound}(q) \pitchfork v = \emptyset\ \textbf{and}\ \mathsf{Bound}(r) \pitchfork u = \emptyset\ \textbf{and}\ v^\sharp \neq u^\sharp \end{bmatrix}\ \textbf{implies}\ \begin{bmatrix} ?(\mathsf{isol}(v\,,\,q) \mid \overline{\mathsf{isol}}(u\,,\,r)) \simeq \delta\ \textbf{and} \\ ?((\mathsf{isol}(v\,,\,q) \cdot q') \mid (\overline{\mathsf{isol}}(u\,,\,r) \cdot r')) \simeq \delta \end{bmatrix}$$

See [28, Appendix C, page 76], for a detailed proof.

Proposition 7 *(Composed* isol- *and* $\overline{\mathsf{isol}}$*-processes cause deadlock, II).*

$$\begin{bmatrix} q\,,\,r \in \mathsf{Basic}\ \textbf{and}\ q\,,\,r \in \mathsf{TauFree}\ \textbf{and}\ \mathsf{Act}(q)\,,\,\mathsf{Act}(r) \subseteq \mathsf{dom}(\Xi) \\ \textbf{and}\ \mathsf{Bound}(q) \pitchfork we = \emptyset\ \textbf{and}\ \mathsf{Bound}(r) \pitchfork wf = \emptyset\ \textbf{and}\ e \neq f \end{bmatrix}\ \textbf{implies}\ \begin{bmatrix} ?(\mathsf{isol}(we\,,\,q) \mid \overline{\mathsf{isol}}(wf\,,\,r)) \simeq \delta\ \textbf{and} \\ ?((\mathsf{isol}(we\,,\,q) \cdot q') \mid (\overline{\mathsf{isol}}(wf\,,\,r) \cdot r')) \simeq \delta \end{bmatrix}$$

See [28, Appendix C, page 79], for a detailed proof.

Although its proof follows the same structure as the proofs of the previous three propositions, we mention Proposition 8 separately for two reasons. First, this proposition does not really generalize a proposition from the previous subsection; second, this proposition plays a crucial role in the proof of an important lemma, Lemma 1, in Section 5.3. Proposition 8 states that if we compose a (co)isolated process $\widehat{\mathsf{isol}}(w\,,\,p)$ using $\between$ with any other process, deadlock occurs.

Proposition 8 *($\widehat{\mathsf{isol}}$-processes cause deadlock in $\between$-ed (left-merged) terms).*

$$\begin{bmatrix} p \in \mathsf{Basic}\ \textbf{and}\ p \in \mathsf{TauFree}\ \textbf{and} \\ \mathsf{Act}(p) \subseteq \mathsf{dom}(\Xi)\ \textbf{and}\ \mathsf{Bound}(p) \pitchfork w = \emptyset \end{bmatrix}\ \textbf{implies}\ \begin{bmatrix} ?(\widehat{\mathsf{isol}}(w\,,\,p) \between q) \simeq \delta\ \textbf{and} \\ ?((\widehat{\mathsf{isol}}(w\,,\,p) \cdot p') \between q) \simeq \delta \end{bmatrix}$$

See [28, Appendix C, page 81], for a detailed proof.

5.3. Synchronization and preservation

We proceed with a series of more significant properties that concern *synchronization* and *preservation*, starting with the former.

[15] Alternatively, we could define a function toSet for converting strings to sets and require $\mathsf{Bound}(p) \cap \mathsf{toSet}(w) = \emptyset$. We favor the $\pitchfork$-notation, because it requires a bit less space, especially in proofs.

Synchronization Lemma 1 states that the parallel composition operator, when operating on the isolation and the coisolation of the same process, behaves as the synchronous composition operator. Intuitively, this lemma captures the phenomenon that (co)isolated processes execute in lockstep when appropriately synchronized by ?: when composed in parallel, an isolated processes and its coisolated sibling always wait for each other until they can perform an auxiliary action and its dual together.

Lemma 1 *(Synchronization lemma).*

$$\big[p \in \mathsf{Basic}\ \textbf{and}\ p \in \mathsf{TauFree}\ \textbf{and}\ \mathsf{Act}(p) \subseteq \mathsf{dom}(\Xi)\ \textbf{and}\ \mathsf{Bound}(p) \pitchfork w = \emptyset\big]$$
$$\textbf{implies}\ \begin{bmatrix} ?(\mathsf{isol}(w\,,\,p) \parallel \overline{\mathsf{isol}}(w\,,\,p)) \simeq ?(\mathsf{isol}(w\,,\,p) \mid \overline{\mathsf{isol}}(w\,,\,p))\ \textbf{and} \\ ?((\mathsf{isol}(w\,,\,p)\cdot p') \parallel (\overline{\mathsf{isol}}(w\,,\,p)\cdot \overline{p}')) \simeq ?((\mathsf{isol}(w\,,\,p)\cdot p') \mid (\overline{\mathsf{isol}}(w\,,\,p)\cdot \overline{p}')) \end{bmatrix}$$

Proof (sketch). By Axiom M, the parallel composition of $\mathsf{isol}(w\,,\,p)$ and $\overline{\mathsf{isol}}(w\,,\,p)$ is provably equal to a nondeterministic choice among three options. The first two options have the shape $\mathsf{isol}(w\,,\,p) \between q$ and $\overline{\mathsf{isol}}(w\,,\,p) \between q$. Distribute ? over + by Axiom Q3, and apply Proposition 8 to conclude that those first two options are provably equal to δ (derive the premise of Proposition 8 from the premise of this lemma). After eliminating these δ-s by Axiom A6, only the third option of the choice remains, which completes the proof.

See [28, Appendix D, page 82], for a detailed proof. □

Preservation The remaining four lemmas in this subsection concern properties stating that the basic operators of the algebra used are preserved by split (i.e., split is homomorphic with respect to the basic operators). These properties make the proof of correctness in Section 5.4 relatively straightforward, but in some sense move the main proof obligations (and complexities) to the lemmas in this subsection.

We start with Lemma 2, which states that + is preserved by split (i.e., split is homomorphic[16] with respect to +).

Lemma 2 *(Preservation lemma for +).*

$$\begin{bmatrix} q+r \in \mathsf{Basic}\ \textbf{and}\ q+r \in \mathsf{TauFree}\ \textbf{and} \\ \mathsf{Act}(q+r) \subseteq \mathsf{dom}(\Xi)\ \textbf{and}\ \mathsf{Bound}(q+r) \pitchfork w = \emptyset \end{bmatrix} \textbf{implies}\ \mathsf{split}(w\,,\,q+r) \simeq \mathsf{split}(w1\,,\,q) + \mathsf{split}(w2\,,\,r)$$

Proof (sketch). By the definition of split and by Lemma 1 (derive the premise of Lemma 1 from the premise of this lemma), conclude that $\mathsf{split}(w\,,\,q+r)$ is provably equal to $?(\mathsf{isol}(w\,,\,q+r) \mid \overline{\mathsf{isol}}(w\,,\,q+r))$. Apply the definitions of isol and $\overline{\mathsf{isol}}$ to obtain $?((q_1+r_1) \mid (q_2+r_2))$ for $q_1 = \mathsf{isol}(w1\,,\,q)$, $r_1 = \mathsf{isol}(w2\,,\,r)$, $q_2 = \overline{\mathsf{isol}}(w1\,,\,q)$, $r_2 = \overline{\mathsf{isol}}(w2\,,\,r)$. Distribute | over + by Axiom S7, and afterwards, distribute ? over + by Axiom Q3. This yields the process $?(q_1 \mid q_2) + ?(q_1 \mid r_2) + ?(r_1 \mid q_2) + ?(r_1 \mid r_2)$. The alternative composition of the first and the last option give the required result (after applying Lemma 1 to each). To get rid of the middle two options, conclude that both of them are provably equal to δ by Proposition 6 (derive the premise of Proposition 6 for both of them from the premise of this lemma), and eliminate them by Axiom A6.

See [28, Appendix D, page 83], for a detailed proof. □

We continue with Lemma 3, which states that $\rightarrowtriangle$, $\diamond$ is preserved by split (i.e., split is homomorphic[16] with respect to $\rightarrowtriangle$, $\diamond$).

Lemma 3 *(Preservation lemma for $\rightarrowtriangle$, $\diamond$).*

$$\mathsf{split}(w\,,\,c \rightarrowtriangle q \diamond r) \simeq c \rightarrowtriangle \mathsf{split}(w1\,,\,q) \diamond \mathsf{split}(w2\,,\,r)$$

Proof (sketch). Distinguish two cases: $c \approx \mathit{true}$ and $c \approx \mathit{false}$. In the former case, by the definition of split, isol and $\overline{\mathsf{isol}}$, and c, conclude that $\mathsf{split}(w\,,\,c \rightarrowtriangle q \diamond r)$ is provably equal to $?((\mathit{true} \rightarrowtriangle \mathsf{isol}(w1\,,\,q) \diamond r') \mid (\mathit{true} \rightarrowtriangle \overline{\mathsf{isol}}(w1\,,\,q) \diamond r''))$. Reduce these processes by Axiom COND1 (from left to right) and apply split to obtain $\mathsf{split}(w1\,,\,q)$. Use Axiom COND1 once more (from right to left this time) to get the required result. The other case follows analogously.

See [28, Appendix D, page 85], for a detailed proof. □

[16] With abuse of terminology, ignoring that w becomes $w1$ and $w2$.

The following lemma, Lemma 4, states that $\sum$ is preserved by split (i.e., split is homomorphic[16] with respect to $\sum$), if the domain of quantification has only finitely many elements. We require finiteness, because otherwise we cannot apply Proposition 1 in the proof, which we do.[17]

Lemma 4 *(Preservation lemma for $\sum$).*

$$\left[\begin{array}{c}\sum_{x\in\{d_1,\ldots,d_\ell\}} q \in \mathsf{Basic} \textbf{ and } \sum_{x\in\{d_1,\ldots,d_\ell\}} q \in \mathsf{TauFree} \textbf{ and} \\ \mathsf{Act}(\sum_{x\in\{d_1,\ldots,d_\ell\}} q) \subseteq \mathsf{dom}(\Xi) \textbf{ and } \mathsf{Bound}(\sum_{x\in\{d_1,\ldots,d_\ell\}} q) \pitchfork w = \emptyset\end{array}\right] \textbf{ implies } \mathsf{split}(w, \sum_{x\in\{d_1,\ldots,d_\ell\}} q) \simeq \sum_{x\in\{d_1,\ldots,d_\ell\}} \mathsf{split}(wx, q)$$

Proof (sketch). By the definition of split and by Lemma 1 (derive the premise of Lemma 1 from the premise of this lemma), conclude that $\mathsf{split}(w, \sum_{x\in\{d_1,\ldots,d_\ell\}} q)$ is provably equal to $?(\mathsf{isol}(w, \sum_{x\in\{d_1,\ldots,d_\ell\}} q) \mid \overline{\mathsf{isol}}(w, \sum_{x\in\{d_1,\ldots,d_\ell\}} q))$. Then apply Proposition 1, from left to right, to obtain the same process but with an ordinary alternative composition: $?(\sum_{i=1}^{\ell} \mathsf{isol}(wd_i, q[d_i/x]) \mid \sum_{i=1}^{\ell} \overline{\mathsf{isol}}(wd_i, q[d_i/x]))$.[18] Distribute $\mid$ over $+$ by Axiom S7, and afterwards, distribute ? over $+$ by Axiom Q3. This yields the process $\sum_{i=1}^{\ell}\sum_{j=1}^{\ell} ?(\mathsf{isol}(wd_i, q[d_i/x]) \mid \overline{\mathsf{isol}}(wd_j, q[d_j/x]))$. The alternative composition of the processes on the "diagonal" yields the desired result (after applying Q3 and Proposition 1, from right to left). To get rid of the processes *not* on the diagonal, conclude that each of them is provably equal to δ by Proposition 7 (derive the premise of Proposition 7 for each of them from the premise of this lemma), and eliminate them by Axiom A6.

See [28, Appendix D, page 85], for a detailed proof. □

The final lemma of this subsection states that $\cdot$ is preserved by split (i.e., split is homomorphic[16] with respect to $\cdot$). The proof of Lemma 5 requires the application of the other preservation lemmas and, in contrast to those lemmas, involves structural induction. This makes Lemma 5 the most complex among the lemmas in this subsection.

Lemma 5 *(Preservation lemma for $\cdot$).*

$$\left[\begin{array}{c} q\cdot r \in \mathsf{Basic} \textbf{ and } q\cdot r \in \mathsf{TauFree} \textbf{ and} \\ \mathsf{Act}(q\cdot r) \subseteq \mathsf{dom}(\Xi) \textbf{ and } \mathsf{Bound}(q\cdot r) \pitchfork w = \emptyset\end{array}\right] \textbf{ implies } \mathsf{split}(w, q\cdot r) \simeq \mathsf{split}(w1, q)\cdot\mathsf{split}(w2, r)$$

Proof (sketch). By the definition of split and by Lemma 1 (derive the premise of Lemma 1 from the premise of this lemma), conclude that $\mathsf{split}(w, q\cdot r)$ is provably equal to $?(\mathsf{isol}(w, q\cdot r) \mid \overline{\mathsf{isol}}(w, q\cdot r))$. Apply the definitions of isol and $\overline{\mathsf{isol}}$ to obtain $?((q_1\cdot r_1) \mid (q_2\cdot r_2))$ for $q_1 = \mathsf{isol}(w1, q)$, $r_1 = \mathsf{isol}(w2, r)$, $q_2 = \overline{\mathsf{isol}}(w1, q)$, $r_2 = \overline{\mathsf{isol}}(w2, r)$. Then, proceed by induction on the structure of q to show that $?((q_1\cdot r_1) \mid (q_2\cdot r_2))$ is provably equal to $\mathsf{split}(w1, q)\cdot ?(r_1 \parallel r_2)$ (afterwards, the required result follows straightforwardly by identifying $\mathsf{split}(w2, r)$ with $?(r_1 \parallel r_2)$).

Establish the base of the induction (q is a multiaction or δ) by applying Axiom S6, Axiom Q4, and Lemma 1. To prove the inductive step, set up a case distinction for the main operator of q. Cases $+$, $\rightarrowtriangle$, $\diamond$, and $\sum$ follow by similar reasoning as in Lemmas 2, 3, and 4. The key difference between those lemmas and the corresponding cases in the inductive step lies in the presence of r_1 and r_2 in the latter. Using the induction hypothesis and the grayed out consequents of Propositions 6 (for $+$) and 7 (for $\sum$), one can "neutralize" their effect and, basically, follow the same structure as the proofs of the other preservation lemmas. For proving the $\cdot$ case, the induction hypothesis and Lemma 1 suffice.

See [28, Appendix D, page 93], for a detailed proof. □

[17] We speculate that generalizing Lemma 4 to processes involving infinite summation is possible. Instead of using Proposition 1, the idea is to more directly prove that for any possibly infinite D, $\mathsf{split}(w, \sum_{x\in D} q)$ is equal to:

$$\begin{array}{rl} & \sum_{x\in D}\sum_{y\in D\setminus\{x\}} x\approx y \rightarrowtriangle ?(\mathsf{isol}(wx, q) \mid \overline{\mathsf{isol}}(wy, q[y/x]))\diamond\delta \\ + & \sum_{x\in D}\sum_{y\in\{x\}} x\approx y \rightarrowtriangle ?(\mathsf{isol}(wx, q) \mid \overline{\mathsf{isol}}(wy, q[y/x]))\diamond\delta \end{array}$$

(Roughly: expand the definition of split, apply Axiom SUM2 to rename x to y in the coisolated process, work the summation operators to the left, divide the inner summation over two outer summations as above, and finally, show that unless x and y are equal, the appropriately synchronized composition of $\mathsf{isol}(wx, q)$ and $\overline{\mathsf{isol}}(wy, q[y/x])$ deadlocks.) Once in this form, the rest of the proof is similar to the last part of our current proof for finite summations.

However, even after generalizing Lemma 4 in this way, we still cannot claim that our splitting procedure works for processes involving infinite summation, because we use the assumed finiteness of summation also in our current proofs of Propositions 2, 3, and 4. Although we strongly believe those propositions to hold also for infinite summation, we have yet to find a suitable proof technique for formally establishing this: what we tried so far became notationally so extremely complex that we decided to resort to the finiteness assumption. We leave this and the details of generalizing Lemma 4 for future work.

[18] Actually, the application of Proposition 1 yields $?(\sum_{i=1}^{\ell} \mathsf{isol}(wx, q)[d_i/x] \mid \sum_{i=1}^{\ell} \overline{\mathsf{isol}}(wx, q)[d_i/x])$. However, one can show that this is provably equal to $?(\sum_{i=1}^{\ell} \mathsf{isol}(wd_i, q[d_i/x]) \mid \sum_{i=1}^{\ell} \overline{\mathsf{isol}}(wd_i, q[d_i/x]))$ by induction on the structure of q.

5.4. Correctness

Next, we state three theorems which, in increasing level of generality, establish the correctness of our splitting procedure. The first theorem, Theorem 1, states that a split multiaction has the same behavior as the original, unsplit multiaction.

Theorem 1 *(Correctness theorem for multiactions).*

$$\left[\alpha \in \mathsf{TauFree} \ \textbf{and} \ \mathsf{Act}(\alpha) \subseteq \mathsf{dom}(\Xi)\right] \ \textbf{implies} \ \mathsf{split}(w\,,\,\alpha) \simeq \alpha$$

Proof (sketch). By the definition of split, Lemma 1 (derive the premise of Lemma 1 from the premise of this lemma), and Axiom SMA, conclude that $\mathsf{split}(w\,,\,\alpha)$ is provably equal to $?(\mathsf{isol}(w\,,\,\alpha) \sqcup \overline{\mathsf{isol}}(w\,,\,\alpha))$. Then, by straightforward induction on the structure of α, establish:

- α is provably equal to $\bigsqcup_{i=1}^{n} a_i(\mathbf{d}_i) \sqcup \bigsqcup_{i=1}^{n'} a'_i(\mathbf{d}'_i)$
- $\mathsf{isol}(w\,,\,\alpha)$ is provably equal to $\bigsqcup_{i=1}^{n} (a_i(\mathbf{d}_i) \sqcup \xi_{w^\sharp}(a_i)(w^\flat)) \sqcup \bigsqcup_{i=1}^{n'} \bar{\xi}_{w^\sharp}(a'_i)(w^\flat)$
- $\overline{\mathsf{isol}}(w\,,\,\alpha)$ is provably equal to $\bigsqcup_{i=1}^{n'} (a'_i(\mathbf{d}'_i) \sqcup \xi_{w^\sharp}(a'_i)(w^\flat)) \sqcup \bigsqcup_{i=1}^{n} \bar{\xi}_{w^\sharp}(a_i)(w^\flat)$

Insert the latter two results in $?(\mathsf{isol}(w\,,\,\alpha) \sqcup \overline{\mathsf{isol}}(w\,,\,\alpha))$, and by Axiom MA2, rearrange the actions in the resulting multiaction to obtain:

$$?\left(\bigsqcup_{i=1}^{n} (a_i(\mathbf{d}_i) \sqcup \xi_{w^\sharp}(a_i)(w^\flat) \sqcup \bar{\xi}_{w^\sharp}(a_i)(w^\flat)) \sqcup \bigsqcup_{i=1}^{n'} (a'_i(\mathbf{d}'_i) \sqcup \xi_{w^\sharp}(a'_i)(w^\flat) \sqcup \bar{\xi}_{w^\sharp}(a'_i)(w^\flat))\right)$$

Then, because ? effectively filters out all pairs of an auxiliary action and its dual (e.g., $\xi_{w^\sharp}(a_i)(w^\flat)$ and $\bar{\xi}_{w^\sharp}(a_i)(w^\flat)$), without affecting the original actions (because the sets of auxiliary and original actions do not overlap by Definition 3), obtain $\bigsqcup_{i=1}^{n} a_i(\mathbf{d}_i) \sqcup \bigsqcup_{i=1}^{n'} a'_i(\mathbf{d}'_i)$, which is provably equal to α (by the first item in the above itemization).
See [28, Appendix E, page 94], for a detailed proof. □

The following theorem states that a split basic process has the same behavior as the original, unsplit process.

Theorem 2 *(Correctness theorem for basic processes).*

$$\left[\begin{array}{c} p \in \mathsf{Basic} \ \textbf{and} \ p \in \mathsf{TauFree} \ \textbf{and} \\ \mathsf{Act}(p) \subseteq \mathsf{dom}(\Xi) \ \textbf{and} \ \mathsf{Bound}(p) \pitchfork w = \emptyset \end{array}\right] \ \textbf{implies} \ \mathsf{split}(w\,,\,p) \simeq p$$

Proof (sketch). Prove this theorem by a relatively straightforward induction on the structure of p. The base case (p is a multiaction or δ) follows immediately from Theorem 1 (derive the premise of Theorem 1 from the premise of this theorem) or the definition of split (for δ). To prove the inductive step, set up a case distinction for the main operator of p, and prove those cases quickly using the preservation lemmas (derive the premises of Lemmas 2, 3, 4, and 5 from the premise of this theorem). For example ($p = q + r$):

$$\mathsf{split}(w\,,\,p) = \mathsf{split}(w\,,\,q+r) \overset{\text{Lemma 2}}{\simeq} \mathsf{split}(w1\,,\,q) + \mathsf{split}(w2\,,\,r) \overset{\text{IH}}{\simeq} q + r = p$$

See [28, Appendix E, page 96], for a detailed proof. □

The last theorem of this paper states that split process definitions (potentially mutually recursive) have the same behavior as the original, unsplit process definitions. To prove this theorem, we find it helpful to work with single recursive process definitions instead of collections of mutually recursive ones (because the former allows for a straightforward application of RSP as explained in Section 2.3). To do this without resorting to weaker results, we first present a proposition about the untimed subset of mCRL2, adapted from [47]: Proposition 9 states that one can collapse k, potentially mutually recursive, process definitions (referenced by $P_1\,,\,\ldots\,,\,P_k$) into a single process definition (referenced by $\widetilde{P}$).

$$
\begin{array}{ll}
\mathsf{Ref}(\boldsymbol{\alpha}) & = \emptyset \\
\mathsf{Ref}(P(\mathbf{d})) & = \{P\} \\
\mathsf{Ref}(q \oplus r)\,,\ \mathsf{Ref}(c \mathrel{-\!\!\triangleright} q \diamond r) & = \mathsf{Ref}(q) \cup \mathsf{Ref}(r) \\
\mathsf{Ref}(\textstyle\sum_{x \in D} q)\,,\ \mathsf{Ref}(f(q)) & = \mathsf{Ref}(q)
\end{array}
$$

Fig. 17. Definition of Ref.

Proposition 9. *(See [47, Section 4.3].)*

$$
\begin{bmatrix}
P_1(\mathbf{x}_1 : \mathbf{D_1}) = p_1\,, \\
\vdots \\
P_k(\mathbf{x}_k : \mathbf{D}_k) = p_k\,, \\
\widetilde{P}(y\,,\ \mathbf{x} : \mathbb{N} \times \mathbf{D}) = y \approx 1 \mathrel{-\!\!\triangleright} p_1[P_1(\mathbf{d}) := \widetilde{P}(1\,,\ h(\mathbf{d}))] \cdots [P_k(\mathbf{d}) := \widetilde{P}(k\,,\ h(\mathbf{d}))] \diamond \\
\qquad\qquad \vdots \\
\qquad\qquad y \approx k \mathrel{-\!\!\triangleright} p_k[P_1(\mathbf{d}) := \widetilde{P}(1\,,\ h(\mathbf{d}))] \cdots [P_k(\mathbf{d}) := \widetilde{P}(k\,,\ h(\mathbf{d}))] \diamond \delta \\
\qquad\qquad\quad \textbf{and}\ \ h = \mathsf{harmonizer}(\mathbf{D}_1 \cup \cdots \cup \mathbf{D}_k\,,\ \mathbf{D}) \\
\qquad\qquad\quad \textbf{implies}\ \ \left[P_i \simeq \widetilde{P}(i)\ \ \textbf{for all}\ \ 1 \leqslant i \leqslant k\right]
\end{bmatrix}
$$

Proposition 9 may look complex, but conceptually, it states a rather simple property. Essentially, it corresponds to the "collapsing into one equation" step of the mCRL2 linearization process [47], as follows. Reference $\widetilde{P}$ has a parameter y which represents the indices of the k processes. The body of $\widetilde{P}$ contains a conditional choice dependent on the value of y: if y equals some index i, the body of $\widetilde{P}$ behaves as the body of P_i. Thus: $\widetilde{P}(i) \simeq P_i$. To ensure also that $\widetilde{P}$ contains only references to itself, one should substitute occurrences of $P_1\,,\ \ldots\,,\ P_k$ with $\widetilde{P}$ in p_i. To this end, we write $p_i[P_j(\mathbf{d}) := P(j\,,\ h(\mathbf{d}))]$ for the process resulting from replacing $P_j(\mathbf{d})$ by $P(j\,,\ h(\mathbf{d}))$ in p_i (for any $\mathbf{d}$), for some *harmonization function* h. Such a function maps data tuples in $\mathbf{D}_1 \cup \cdots \cup \mathbf{D}_k$ to data tuples in $\mathbf{D}$. Intuitively, h transforms the parameters of each of the process references $P_1\,,\ \ldots\,,\ P_k$ to a single tuple of parameters for $\widetilde{P}$. Neither the precise meaning of harmonization nor the definition of harmonizer matter in the remainder, so we skip them (details appear elsewhere [47]).

We proceed with our final theorem. Let $\mathsf{Ref}(p)$ (defined in Fig. 17) denote the set of references occurring in p.

Theorem 3 *(Correctness theorem for process specifications).*

$$
\begin{bmatrix}
P_1(\mathbf{x}_1 : \mathbf{D_1}) = p_1\,,\ P_1^\dagger(\mathbf{x}_1 : \mathbf{D_1}) = \mathsf{split}(\epsilon\,,\ p_1)\,, \\
\vdots \\
P_k(\mathbf{x}_k : \mathbf{D}_k) = p_k\,,\ P_k^\dagger(\mathbf{x}_k : \mathbf{D_k}) = \mathsf{split}(\epsilon\,,\ p_k) \\
\qquad\qquad \textbf{and}\ \ p_1\,,\ \ldots\,,\ p_k \in \mathsf{TauFree} \\
\qquad\quad \textbf{and}\ \ \mathsf{Act}(p_1)\,,\ \ldots\,,\ \mathsf{Act}(p_k) \subseteq \mathsf{dom}(\Xi)\ \ \textbf{and} \\
\qquad\quad \left[\mathsf{Ref}(p_i) \subseteq \{P_1\,,\ \ldots\,,\ P_k\}\ \ \textbf{for all}\ \ 1 \leqslant i \leqslant k\right]
\end{bmatrix}
\ \ \textbf{implies}\ \ \begin{bmatrix} P_i \simeq P_i^\dagger\ \ \textbf{for all} \\ 1 \leqslant i \leqslant k \end{bmatrix}
$$

Proof (sketch). Apply Proposition 9 to collapse the definitions referenced by $P_1\,,\ \ldots\,,\ P_k$ into one definition $\widetilde{P} = p$. Similarly, apply Proposition 9 to collapse the definitions referenced by $P_1^\dagger\,,\ \ldots\,,\ P_k^\dagger$ into one definition $\widetilde{P}^\dagger = p^\dagger$. To obtain the desired result, show that $\widetilde{P}$ is provably equal to $\widetilde{P}^\dagger$ by demonstrating that some process operator Φ has both $\widetilde{P}$ and $\widetilde{P}^\dagger$ as fixed points (and apply RSP). Define $\Phi(Z) = p[\widetilde{P} := Z]$, and immediately conclude $\Phi(\widetilde{P}) \simeq \widetilde{P}$. To show that also $\Phi(\widetilde{P}^\dagger) \simeq \widetilde{P}^\dagger$, essentially, it suffices to show that $p_i \simeq \mathsf{split}(\epsilon\,,\ p_i)$. This follows from Theorem 2 (derive the premise of Theorem 2 from the premise of this theorem).

See [28, Appendix E, page 101], for a detailed proof. We establish the $\Phi(\widetilde{P}^\dagger) \simeq \widetilde{P}^\dagger$ step with a separate auxiliary theorem [28, Appendix E, Theorem 4, page 99]. □

6. An application of the splitting procedure: splitting connectors

Up to now, we have defined a splitting procedure for untimed mCRL2 and proved its correctness, all independent of Reo. Now, as one of its applications, we use this splitting procedure to justify the region-based optimization technique for Reo implementations (i.e., the version with direct transportation of data and control information in asynchronous regions—see Section 1). First, we formalize (a)synchronous regions in terms of process algebra. Afterwards, we split (process algebraic semantic specifications of) connectors.

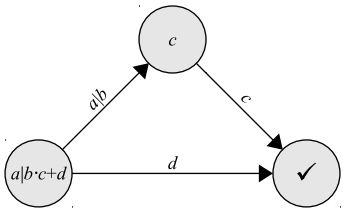


Fig. 18. Example transition system for $p = a \sqcup b \cdot c + d$. The ✓-state represents successful termination.

6.1. Formalization of (a)synchronous regions

We provide a formal definition of the synchronous regions of a connector, based on the mCRL2 semantics of Reo. Let p denote a process describing the behavior of a Reo connector, and let $\longrightarrow$ denote its transition relation (labeled with multiactions).[19] Recall that every action in p represents a channel end or a node end. Let $a \in \mathsf{Act}(p)$ denote one such end. We define the a-synchronous region of p as the smallest set $X_a \subseteq \mathsf{Act}(p)$ such that:

- $a \in X_a$
- $b \in X_a \Rightarrow [\mathsf{Act}(\beta) \subseteq X_a$ **for all** β **such that** $[q \xrightarrow{\beta} q'$ **and** $b \in \mathsf{Act}(\beta)]]$[20]
- $b \in X_a \Rightarrow [\mathsf{Act}(\beta') \subseteq X_a$ **for all** β, β' **such that** $\left[\begin{array}{c} q \xrightarrow{\beta} q' \text{ and } q \xrightarrow{\beta'} q'' \\ \text{and } b \in \mathsf{Act}(\beta) \end{array}\right]]$

The second rule states that all the ends that occur in the same multiaction belong to the same synchronous region. The third rule states that all the ends that can have flow in some state q, but possibly in different transitions leaving q, belong to the same synchronous region. In that case, channel ends may exclude each other from flow, which requires them to synchronize and communicate about their behavior.

To exemplify the previous definition, consider the connector modeled by the process $p = a \sqcup b \cdot c + d$ (we abstract away from data in this example, because data do not influence the regions of the connector). Fig. 18 shows its transition system. Informally, either this connector has flow through a and b followed by flow through c, or it has flow through d. We construct its a-synchronous region starting from the singleton set $X_a = \{a\}$ (first rule). Subsequently, due to the presence of multiaction $a \sqcup b$, we add b to this set (second rule). The transition system of p contains a state with two outgoing transitions: one labeled by $a \sqcup b$, the other labeled by d. Hence, because $a \in X_a$, we add d to X_a (third rule). This concludes the construction: $X_a = X_b = X_d = \{a, b, d\}$ and $X_c = \{c\}$.

We define the set of the synchronous regions of the connector modeled by p as

$$\mathcal{X} = \{X_a \mid a \in \mathsf{Act}(p)\}$$

and the set containing its asynchronous regions as

$$\mathcal{Y} = \{\langle a, b\rangle \mid \mathit{connected}(a, b) \text{ and } a \in X \text{ and } b \in X' \text{ and } X \neq X' \text{ and } X, X' \in \mathcal{X}\}$$

where $\mathit{connected}(a, b)$ holds iff ends a and b belong to the same channel.

6.2. Splitting connectors

As motivated in Section 1, we set out to establish the soundness of splitting connectors along the boundaries of their (a)synchronous regions. However, we can split any (syntactically τ-free) process along any set of actions $\mathbb{A}$ by Theorem 3. This suggests that regardless of its (a)synchronous regions, one can split a connector in any possible way *and* preserve its original semantics. While true in theory, there is a catch for implementations of split connectors in practice: the parallel composition of the isolation and the coisolation of a connector process must *synchronize appropriately*, as represented by the ? operator (see Definition 4). Depending on the particular implementation approach, which in turn may depend on the underlying hardware architecture (see Section 1), performing ? at run-time may cost an unreasonable amount of resources, if possible at all. Next, we explain through examples that *arbitrarily* splitting—despite its theoretical validity—therefore makes no sense in practice (because, as we shall see, the overhead due to communication required for appropriately synchronizing split processes can make running an original, unsplit connector more attractive), whereas splitting based on (a)synchronous regions does make sense.

[19] We have not given the definition of the transition relation (although examples appear in Figs. 8, 18, and 20), because the precise definition does not matter in this paper. See Groote et al. [20].

[20] Square brackets for readability.

Region-based splitting We start with an example of splitting based on (a)synchronous regions. Suppose that we split $\mathsf{fifo1}\langle a, b\rangle$ into two parts: one part contains only a, while the other part contains only b. Recall from Section 3 that the semantics of this channel is given by the process definition $\mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle = \sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\cdot b(x))\cdot \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle$. Splitting along $\mathbb{A} = \{a\}$ (or equivalently, along $\mathbb{A} = \{b\}$) yields:

$$
\begin{aligned}
& \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& \mathsf{split}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\cdot b(x))\cdot \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle) \\
= \;& \mathsf{split}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\cdot b(x)))\cdot \mathsf{split}(\epsilon\,,\, \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle) \\
= \;& ?(\mathsf{isol}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\cdot b(x)))\parallel \overline{\mathsf{isol}}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\cdot b(x))))\cdot \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& ?(\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}\mathsf{isol}(x\,,\, a(x)\cdot b(x))\parallel \sum_{x\in\mathbb{D}\mathsf{ata}}\overline{\mathsf{isol}}(x\,,\, a(x)\cdot b(x)))\cdot \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& ?(\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\mathsf{isol}(x1\,,\, a(x))\cdot \mathsf{isol}(x2\,,\, b(x)))\parallel \\
& \quad \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\overline{\mathsf{isol}}(x1\,,\, a(x))\cdot \overline{\mathsf{isol}}(x2\,,\, b(x))))\cdot \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& ?(\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\sqcup \xi_1(a)(x)\cdot \quad \bar{\xi}_2(b)(x))\parallel \\
& \quad \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\quad \bar{\xi}_1(a)(x)\cdot b(x)\sqcup \xi_2(b)(x)))\cdot \mathtt{Fifo1}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger}
\end{aligned}
$$

Here, ? in fact represents the asynchronous region of $\mathsf{fifo1}\langle a; b\rangle$, because it synchronizes the two synchronous regions $\{a\}$ and $\{b\}$. The fact that auxiliary actions happen at the same time as the corresponding original actions represents direct transportation of data and control information in asynchronous regions (see Section 1).

Suppose that we want to implement $p = \sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\sqcup \xi_1(a)(x)\cdot \bar{\xi}_2(b)(x))$ and $q = \sum_{x\in\mathbb{D}\mathsf{ata}}(\bar{\xi}_1(a)(x)\cdot b(x)\sqcup \xi_2(b)(x))$ such that, when run in parallel, they behave as $\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\cdot b(x))$. Crucially, these implementations should perform the synchronization implied by ?. Recall from Section 4 that intuitively, $\xi_1(a)$ represents the act of "disseminating the performance of a," while $\bar{\xi}_1(a)$ represents the act of "discovering the performance of a." Thus, the implementation of p should: (1) accept data x on a and disseminate this acceptance, and (2) discover the dispersal of x on b. Meanwhile, the implementation of q should: (1) discover the acceptance of data x on a, and (2) dispense x on b and disseminate this dispersal. Thus, in each step, the implementations of p and q require only unidirectional communication about their behavior to synchronize: first, the implementation of p performs $\xi_1(a)(x)$ and the implementation of q takes notice of this (by performing $\bar{\xi}_1(a)(x)$); afterwards, p and q switch roles to perform $\bar{\xi}_2(b)(x)$ and $\xi_2(b)(x)$. This shows that different synchronous regions can decide on their behavior independently of each other: region $\{a\}$ does not need to know that region $\{b\}$ will dispense data before it can accept data—it can decide to do so without communication.

We argue that this can yield performance improvements in practice: although the isolation and the coisolation of a process p have the same transition system modulo transition labels (i.e., they have the same syntactic structure), benefits can arise when we compose them in parallel with *another* split process q. In that case, there may exist a transition t of the isolation of p that can proceed independently—without communication among the ends involved—of a transition t' of the coisolation of q. Without splitting, in contrast, communication among the ends involved in t and t' must always take place to decide on whether to behave according to t, t', or both. But in the split case, the ends can act independently. For instance, if we put two split $\mathsf{fifo1}$ instances in sequence (as in Fig. 1a), the source end a of the first $\mathsf{fifo1}$ can proceed independently of the sink end b of the second $\mathsf{fifo1}$. This means that, if empty, the first $\mathsf{fifo1}$ can accept a data item on a (and place it in its buffer) without communicating with b. Similarly, if full, the second $\mathsf{fifo1}$ can dispense a data item on b (and remove it from its buffer) without communicating with a. In contrast, if we put two unsplit $\mathsf{fifo1}$ instances in sequence, the source end a and the sink end b communicate with each other to decide on their joint behavior, even though the behavior of those ends does not depend on each other. By splitting, one avoids this unnecessary communication.

Arbitrary splitting To demonstrate that splitting arbitrarily makes no sense, suppose that we split $\mathsf{sync}\langle a, b\rangle$ into two parts: one part contains only a, while the other part contains only b. Recall from Section 3 that the semantics of this channel is given by the process definition $\mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle = \sum_{x\in\mathbb{D}\mathsf{ata}} a(x)\sqcup b(x)\cdot \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle$. Splitting along $\mathbb{A} = \{a\}$ (or equivalently, along $\mathbb{A} = \{b\}$) yields:

$$
\begin{aligned}
& \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& \mathsf{split}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} a(x)\sqcup b(x)\cdot \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle) \\
= \;& \mathsf{split}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} a(x)\sqcup b(x))\cdot \mathsf{split}(\epsilon\,,\, \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle) \\
= \;& ?(\mathsf{isol}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} a(x)\sqcup b(x))\parallel \overline{\mathsf{isol}}(\epsilon\,,\, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} a(x)\sqcup b(x)))\cdot \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& ?(\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}\mathsf{isol}(x\,,\, a(x)\sqcup b(x))\parallel \sum_{x\in\mathbb{D}\mathsf{ata}}\overline{\mathsf{isol}}(x\,,\, a(x)\sqcup b(x)))\cdot \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& ?(\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\mathsf{isol}(x\,,\, a(x))\sqcup \mathsf{isol}(x\,,\, b(x)))\parallel \\
& \quad \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\overline{\mathsf{isol}}(x\,,\, a(x))\sqcup \overline{\mathsf{isol}}(x\,,\, b(x))))\cdot \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger} \\
= \;& ?(\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(a(x)\sqcup \xi_\epsilon(a)(x)\sqcup \quad \bar{\xi}_\epsilon(b)(x))\parallel \\
& \quad \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\quad \bar{\xi}_\epsilon(a)(x)\sqcup b(x)\sqcup \xi_\epsilon(b)(x)))\cdot \mathtt{Sync}\langle \mathtt{a};\mathtt{b}\rangle^{\dagger}
\end{aligned}
$$

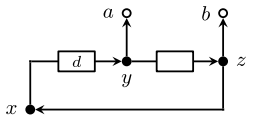


Fig. 19. Sequencer2.

Now, as in the previous example, suppose that we want to implement

$$p = \sum_{x \in \mathbb{D}\mathrm{ata}} \big(a(x) \sqcup \xi_1(a)(x) \sqcup \bar{\xi}_2(b)(x)\big) \quad \text{and} \quad q = \sum_{x \in \mathbb{D}\mathrm{ata}} \big(\bar{\xi}_1(a)(x) \sqcup b(x) \sqcup \xi_2(b)(x)\big)$$

such that, when run in parallel, they behave as $\sum_{x \in \mathbb{D}\mathrm{ata}}(a(x) \cdot b(x))$. As before, these implementations should perform the synchronization implied by ?. Thus, the implementation of p should accept data x on a, disseminate this acceptance, and discover the dispersal of x on b. Meanwhile, the implementation of q should discover the acceptance of data x on a, dispense x on b, and disseminate this dispersal. *All of these actions must occur at the same time*. This means that, in contrast to our previous example, the implementations of p and q must engage in *bi*directional communication with each other about the acceptance of data on a and the dispersal of data on b. This suggests that the two ends of $\mathsf{sync}\langle a, b\rangle$ must synchronize with each other—they belong to the same synchronous region and cannot decide on their behavior independently—making it unreasonable to split them in the first place: the communication necessary to realize the necessary synchronization inflicts overhead, making it more attractive to run the original $\mathsf{sync}\langle a, b\rangle$ without splitting.

Implementation sketch We sketch an implementation of the split $\mathsf{fifo1}\langle a, b\rangle$ on a shared memory machine with multithreading. First, we instantiate two threads, A and B, for the processes $p = \sum_{x \in \mathbb{D}\mathrm{ata}}(a(x) \sqcup \xi_1(a)(x) \cdot \bar{\xi}_2(b)(x))$ and $q = \sum_{x \in \mathbb{D}\mathrm{ata}}(\bar{\xi}_1(a)(x) \cdot b(x) \sqcup \xi_2(b)(x))$. Every multiaction α translates to the atomic execution of a block of code representing the actions occurring in α. We implement the action $\xi_1(a)(x)$ as "write x to a shared memory location" and the action $\bar{\xi}_1(a)(x)$ as "wait until the next read of the memory location returns x" (not the read itself). There exist several ways to implement thread B's waiting: B could use busy-waiting (in which case it repeatedly checks the memory location; in this case, B performs $\bar{\xi}_1(a)(x)$ between two successive reads) or A and B could agree on using a monitor (in which case B waits on a condition variable and A notifies B immediately after its write; in this case, B performs $\bar{\xi}_1(a)(x)$ on awakening). Once B has performed $\bar{\xi}_1(a)(x)$ (after which it knows that A has accepted data on a and put data in the memory location), it can perform the actual read on the memory location, dispense the data on b, and set another shared location for $\xi_2(b)$ and $\bar{\xi}_2(b)$ (implemented symmetrically).

Generally, every connector split based on its (a)synchronous regions can be implemented as sketched above. Whether such an implementation realizes unidirectional communication reasonably efficiently depends on the underlying architecture. On the shared memory machine in our $\mathsf{fifo1}\langle a, b\rangle$ example, the sketched implementation is relatively efficient, because a thread performing an action $\xi_w(a)$ by writing to a memory location knows immediately after completing everything involved in that write (this may include signaling the other thread) that the other thread has performed $\bar{\xi}_w(a)$: from that point onward, the next read of the memory location is guaranteed to see the result of the write, regardless of what threads do until that read.[21] This is a sufficient condition for considering $\bar{\xi}_w(a)$ to be performed (given the interpretation of $\bar{\xi}_w(a)$ in our implementation sketch). In other words, communication is truly unidirectional (i.e., the writing thread requires no acknowledgment of the other thread). In contrast, if the isolation and the coisolation of a split process run on different machines, the machine that performs $\xi_w(a)$ by sending a value to another machine must block until the other machine has acknowledged the receipt of the value. Otherwise, the sending machine has no guarantee that the receiving machine has actually received the value. This is important, because as long as the receiving machine has not received the value, it has not performed $\bar{\xi}_w(a)$. Consequently, if the sending machine does not wait for an acknowledgment before it proceeds, the next action of the sending machine may precede $\bar{\xi}_w(a)$, which violates the process semantics. Waiting for an acknowledgment, however, can be expensive and negatively affects performance. Note also that the acknowledgment is a form of "metacommunication", which makes the implementation of $\xi_w(a)$ and $\bar{\xi}_w(a)$ essentially bidirectional. This is exactly what we try to avoid by splitting based on (a)synchronous regions.

Summarizing, if every auxiliary action and its dual are performed on the same shared memory machine, one can implement the processes containing those actions along the same lines as the implementation sketched above and get reasonable performance. If, in contrast, an auxiliary action and its dual are executed on different machines, the sketched implementation seems unsatisfactory. (Perhaps we need a different splitting procedure for such architectures—see the future work in Section 8.)

6.3. *Example:* Sequencer2

In this section, we further illustrate connector splitting based on their (a)synchronous regions with an example. Fig. 19 shows the Sequencer2 connector, introduced by Arbab [2]. Informally, this connector dispenses data value d first on a, then

[21] Assuming that threads do not overwrite the memory location before the next read.

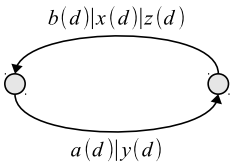


Fig. 20. Transition system of $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow}$.

on b, then on a, then on b, et cetera. One can easily extend this connector to more than two output nodes [2]. Formally, the following process definition, composed from process definitions of channels and nodes, defines the semantics of Sequencer2:

$$\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle = \partial_B\left(\Gamma_C\left(\begin{array}{c} \texttt{B}\langle\widetilde{\texttt{a}}_1\rangle \parallel \texttt{B}\langle\widetilde{\texttt{b}}_1\rangle \\ \texttt{Sync}\langle\texttt{y}_2;\texttt{a}_1\rangle \parallel \texttt{Sync}\langle\texttt{z}_2;\texttt{b}_1\rangle \\ \texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle \parallel \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle \parallel \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle \parallel \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle \parallel \\ \langle\widetilde{\texttt{x}}_1;\widetilde{\texttt{x}}_2\rangle \parallel \texttt{Sync}\langle\texttt{z}_3;\texttt{x}_1\rangle \end{array}\right)\right)$$

The exact definition of B and C do not matter in the rest of this section. (They are defined along the same lines as in the examples at the end of Section 3.) The process reference $\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle$ refers to the process $\texttt{y}_1(d)\cdot\texttt{x}_2(d)\cdot\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle$ (i.e., it represents an initially full fifo1 channel that accepts and dispenses only d). There are two approaches to splitting $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ based on its (a)synchronous regions, both of which yield a split process that is provably equal to the original. We explain these approaches below.

Global approach In the global approach, the aim is to split the whole process into two parallel processes—the "global" isolation and the "global" coisolation—while simplifying all other parallel behavior to sequential behavior (in a semantics-preserving way). Afterward, we can implement those two processes and run them in parallel under appropriate synchronization. This global splitting approach corresponds to the centralized implementation approach to Reo (see Section 1).

First, we compute the (a)synchronous regions based on the labeled transition system of the whole process. To get that transition system, we simplify $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ to a sequential process: using the axioms in Section 2.3, we expand the parallel composition operators, apply the communication rules in C, and block the remaining actions in B to show that $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ is provably equal to $\texttt{a}(d)\sqcup\texttt{y}(d)\cdot\texttt{b}(d)\sqcup\texttt{x}(d)\sqcup\texttt{z}(d)\cdot\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$. Next, we can straightforwardly compute the desired labeled transition system (shown in Fig. 20) and apply the definitions in Section 6.1 to find the following synchronous regions: $\{\texttt{a},\texttt{y}\}$ and $\{\texttt{b},\texttt{x},\texttt{z}\}$. For future reference, we introduce the following process definition:

$$\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow} = \texttt{a}(d)\sqcup\texttt{y}(d)\cdot\texttt{b}(d)\sqcup\texttt{x}(d)\sqcup\texttt{z}(d)\cdot\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow}$$

Using RSP, one can straightforwardly show $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle \simeq \texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow}$.

Given the computed synchronous regions, we proceed by splitting along $\{\texttt{a},\texttt{y}\}$ (or equivalently, along $\{\texttt{b},\texttt{x},\texttt{z}\}$). In fact, we can do so in two different ways: we can split $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$, or we can split $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow}$. Because the processes they refer to are provably equal (and because a split process is provably equal to its original by Theorem 3), whether we split $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ or $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow}$ does not matter as far as provable equality is concerned. However, splitting $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ and afterward simplifying the resulting isolation and coisolation to sequential processes (by parallel expansion, application of communication, and blocking) requires more effort than splitting $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle_{\downarrow}$, as shown below.

We start by splitting and simplifying $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$. First, we modify the set of actions $\{\texttt{a},\texttt{y}\}$ with respect to which we split by "inverting" the communication rules that produced $\texttt{a}$ and $\texttt{y}$ (note that those actions do not occur in the process referred to by $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ but were introduced as part of applying communication rules in C). This gives us the following set $\mathbb{A} = \{\texttt{a}_1,\widetilde{\texttt{a}}_1,\texttt{y}_1,\texttt{y}_2,\texttt{y}_3,\widetilde{\texttt{y}}_1,\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\}$. Splitting with respect to that set yields the following:

$$\begin{array}{ll} & \texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle^{\dagger} \\ \simeq & \mathsf{split}(\epsilon,\ \partial_B(\Gamma_C(\cdots \parallel \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle \parallel \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle \parallel \cdots))) \\ = & \partial_B(\Gamma_C(\cdots \parallel \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle^{\dagger} \parallel \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^{\dagger} \parallel \cdots)) \\ \simeq & \partial\Gamma(\cdots \parallel \mathsf{split}(\epsilon,\ \sum_{x\in\mathbb{D}\mathsf{ata}}\widetilde{\texttt{y}}_1(x)\sqcup\widetilde{\texttt{y}}_2(x)\sqcup\widetilde{\texttt{y}}_3(x)\cdot\texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle) \\ & \qquad \parallel \mathsf{split}(\epsilon,\ \sum_{x\in\mathbb{D}\mathsf{ata}}(\texttt{y}_3(x)\cdot\texttt{z}_1(x))\cdot\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle) \\ & \qquad \parallel \cdots) \\ \simeq & \partial\Gamma(\cdots \parallel (\mathsf{split}(\epsilon,\ \sum_{x\in\mathbb{D}\mathsf{ata}}\widetilde{\texttt{y}}_1(x)\sqcup\widetilde{\texttt{y}}_2(x)\sqcup\widetilde{\texttt{y}}_3(x))\cdot\texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle^{\dagger}) \\ & \qquad \parallel (\mathsf{split}(\epsilon,\ \sum_{x\in\mathbb{D}\mathsf{ata}}(\texttt{y}_3(x)\cdot\texttt{z}_1(x)))\cdot\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^{\dagger}) \\ & \qquad \parallel \cdots) \end{array}$$

For simplicity, we consider only a representative fragment of the whole process in these equations, namely $\cdots \parallel \mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle \parallel \mathtt{Fifo1}\langle y_3; z_1\rangle \parallel \cdots$. For the same reason, we write $\partial_\Gamma(\cdots)$ instead of $\partial_B(\Gamma_C(\cdots))$. For the next step, as in Example 3 in Section 4.3, we introduce the following process definitions:

$$\begin{aligned}
\mathtt{Fifo1}\langle y_3; z_1\rangle^{\ddagger} &= ?(\mathtt{Fifo1}\langle y_3; z_1\rangle^{\S} \parallel \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S}) \\
\mathtt{Fifo1}\langle y_3; z_1\rangle^{\S} &= \mathsf{isol}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3(x)\cdot z_1(x)))\cdot \mathtt{Fifo1}\langle y_3; z_1\rangle^{\S} \\
\overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S} &= \overline{\mathsf{isol}}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3(x)\cdot z_1(x)))\cdot \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S}
\end{aligned}$$

Using RSP, one can prove $\mathtt{Fifo1}\langle y_3; z_1\rangle^{\dagger} \simeq \mathtt{Fifo1}\langle y_3; z_1\rangle^{\ddagger}$ (see Footnote 13). Similarly, we introduce process definitions $\mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\ddagger}$, $\mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}$, and $\overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}$. Applying those definitions yields the following:

$$\begin{aligned}
\simeq\ & \partial_\Gamma(\cdots \parallel (\mathsf{split}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x))\cdot ?(\mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S} \parallel \overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S})) \\
& \quad \parallel (\mathsf{split}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3(x)\cdot z_1(x)))\cdot ?(\mathtt{Fifo1}\langle y_3; z_1\rangle^{\S} \parallel \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S})) \\
& \quad \parallel \cdots)
\end{aligned}$$

Again as in Example 3 in Section 4.3, we use [28, Appendix D, Lemma 6, page 87], to justify the following steps:

$$\begin{aligned}
\simeq\ & \partial_\Gamma(\cdots \parallel ?(\ \ (\mathsf{isol}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x))\cdot \mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \qquad \parallel (\overline{\mathsf{isol}}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x))\cdot \overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S})) \\
& \quad \parallel ?(\ \ (\mathsf{isol}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3(x)\cdot z_1(x)))\cdot \mathtt{Fifo1}\langle y_3; z_1\rangle^{\S}) \\
& \qquad \parallel (\overline{\mathsf{isol}}(\epsilon\,, \textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3(x)\cdot z_1(x)))\cdot \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S})) \\
& \quad \parallel \cdots) \\
=\ & \partial_\Gamma(\cdots \parallel ?(\ \ (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x) \sqcup \xi_\epsilon(\widetilde{y}_1)(x) \sqcup \xi_\epsilon(\widetilde{y}_2)(x) \sqcup \xi_\epsilon(\widetilde{y}_3)(x)\cdot \mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \qquad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \overline{\xi}_\epsilon(\widetilde{y}_1)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_2)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_3)(x)\cdot \overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S})) \\
& \quad \parallel ?(\ \ (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3 \sqcup \xi_1(y_3)(x)\cdot \overline{\xi}_2(z_1)(x))\cdot \mathtt{Fifo1}\langle y_3; z_1\rangle^{\S}) \\
& \qquad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\overline{\xi}_1(y_3)(x)\cdot z_1 \sqcup \xi_2(z_1)(x))\cdot \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S})) \\
& \quad \parallel \cdots)
\end{aligned}$$

Next, we push the ? operators to the top in two steps. The first step is valid by the same argument as used in justifying the last step in Example 3 in Section 4.3 (where we used alphabet axioms—see Footnote 14). The second step follows by the observation that sets B and C (implicit in the notation ∂_Γ as defined above) are disjoint with the sets of actions occurring in the subscripts of the ∂, $\mathcal{T}$, and Γ operators obtained by expanding ?.[22] We thus have the following:

$$\begin{aligned}
\simeq\ & \partial_\Gamma(?(\cdots \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x) \sqcup \xi_\epsilon(\widetilde{y}_1)(x) \sqcup \xi_\epsilon(\widetilde{y}_2)(x) \sqcup \xi_\epsilon(\widetilde{y}_3)(x)\cdot \mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \overline{\xi}_\epsilon(\widetilde{y}_1)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_2)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_3)(x)\cdot \overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3 \sqcup \xi_1(y_3)(x)\cdot \overline{\xi}_2(z_1)(x))\cdot \mathtt{Fifo1}\langle y_3; z_1\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\overline{\xi}_1(y_3)(x)\cdot z_1 \sqcup \xi_2(z_1)(x))\cdot \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S}) \\
& \quad \parallel \cdots)) \\
\simeq\ & ?(\partial_\Gamma(\cdots \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x) \sqcup \xi_\epsilon(\widetilde{y}_1)(x) \sqcup \xi_\epsilon(\widetilde{y}_2)(x) \sqcup \xi_\epsilon(\widetilde{y}_3)(x)\cdot \mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \overline{\xi}_\epsilon(\widetilde{y}_1)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_2)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_3)(x)\cdot \overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3 \sqcup \xi_1(y_3)(x)\cdot \overline{\xi}_2(z_1)(x))\cdot \mathtt{Fifo1}\langle y_3; z_1\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\overline{\xi}_1(y_3)(x)\cdot z_1 \sqcup \xi_2(z_1)(x))\cdot \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S}) \\
& \quad \parallel \cdots))
\end{aligned}$$

By the commutativity and associativity of $\parallel$, we group all the isolation processes and the coisolation processes together as follows:

$$\begin{aligned}
\simeq\ & ?(\partial_\Gamma(\cdots \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \widetilde{y}_1(x) \sqcup \widetilde{y}_2(x) \sqcup \widetilde{y}_3(x) \sqcup \xi_\epsilon(\widetilde{y}_1)(x) \sqcup \xi_\epsilon(\widetilde{y}_2)(x) \sqcup \xi_\epsilon(\widetilde{y}_3)(x)\cdot \mathtt{R}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(y_3 \sqcup \xi_1(y_3)(x)\cdot \overline{\xi}_2(z_1)(x))\cdot \mathtt{Fifo1}\langle y_3; z_1\rangle^{\S}) \\
& \quad \parallel \cdots \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}} \overline{\xi}_\epsilon(\widetilde{y}_1)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_2)(x) \sqcup \overline{\xi}_\epsilon(\widetilde{y}_3)(x)\cdot \overline{\mathtt{R}}\langle \widetilde{y}_1; \widetilde{y}_2, \widetilde{y}_3\rangle^{\S}) \\
& \quad \parallel (\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\overline{\xi}_1(y_3)(x)\cdot z_1 \sqcup \xi_2(z_1)(x))\cdot \overline{\mathtt{Fifo1}}\langle y_3; z_1\rangle^{\S}) \\
& \quad \parallel \cdots))
\end{aligned}$$

Now, just as when we computed the labeled transition system for the whole original process, we simplify the parallel composition of the isolation processes and the coisolation processes (by parallel expansion, application of communication, and blocking). This yields the following:

[22] For this step, we also need four alphabet axioms from [22, Section 5.6]: CD1, CT1, CL2, DT1.

$$\begin{aligned}\simeq\ & ?(\quad (\mathtt{a}(d) \sqcup \mathtt{y}(d) \sqcup \alpha \cdot \qquad\qquad \overline{\beta} \cdot \partial_{\Gamma}(\cdots \parallel \mathtt{R}\langle \widetilde{\mathtt{y}}_1; \widetilde{\mathtt{y}}_2, \widetilde{\mathtt{y}}_3\rangle^{\S} \parallel \mathtt{Fifo1}\langle \mathtt{y}_3; \mathtt{z}_1\rangle^{\S} \parallel \cdots))\\ & \parallel (\qquad\quad \overline{\alpha} \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d) \sqcup \beta \cdot \partial_{\Gamma}(\cdots \parallel \overline{\mathtt{R}}\langle \widetilde{\mathtt{y}}_1; \widetilde{\mathtt{y}}_2, \widetilde{\mathtt{y}}_3\rangle^{\S} \parallel \overline{\mathtt{Fifo1}}\langle \mathtt{y}_3; \mathtt{z}_1\rangle^{\S} \parallel \cdots)))\end{aligned}$$

For:

$$\begin{aligned}\alpha &= \xi_{\epsilon}(\mathtt{a}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{a}}_1)(d) \sqcup \xi_1(\mathtt{y}_1)(d) \sqcup \xi_{\epsilon}(\mathtt{y}_2)(d) \sqcup \xi_1(\mathtt{y}_3)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{y}}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{y}}_2)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{y}}_3)(d)\\ \overline{\alpha} &= \overline{\xi}_{\epsilon}(\mathtt{a}_1)(d) \sqcup \overline{\xi}_{\epsilon}(\widetilde{\mathtt{a}}_1)(d) \sqcup \overline{\xi}_1(\mathtt{y}_1)(d) \sqcup \overline{\xi}_{\epsilon}(\mathtt{y}_2)(d) \sqcup \overline{\xi}_1(\mathtt{y}_3)(d) \sqcup \overline{\xi}_{\epsilon}(\widetilde{\mathtt{y}}_1)(d) \sqcup \overline{\xi}_{\epsilon}(\widetilde{\mathtt{y}}_2)(d) \sqcup \overline{\xi}_{\epsilon}(\widetilde{\mathtt{y}}_3)(d)\\ \beta &= \xi_{\epsilon}(\mathtt{b}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{b}}_1)(d) \sqcup \xi_2(\mathtt{z}_1)(d) \sqcup \xi_{\epsilon}(\mathtt{z}_2)(d) \sqcup \xi_1(\mathtt{z}_3)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{z}}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{z}}_2)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{z}}_3)(d)\\ &\quad \sqcup \xi_{\epsilon}(\mathtt{x}_1)(d) \sqcup \xi_2(\mathtt{x}_2)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{x}}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{x}}_2)(d)\\ \overline{\beta} &= \xi_{\epsilon}(\mathtt{b}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{b}}_1)(d) \sqcup \xi_2(\mathtt{z}_1)(d) \sqcup \xi_{\epsilon}(\mathtt{z}_2)(d) \sqcup \xi_1(\mathtt{z}_3)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{z}}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{z}}_2)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{z}}_3)(d)\\ &\quad \sqcup \xi_{\epsilon}(\mathtt{x}_1)(d) \sqcup \xi_2(\mathtt{x}_2)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{x}}_1)(d) \sqcup \xi_{\epsilon}(\widetilde{\mathtt{x}}_2)(d)\end{aligned}$$

Thus, the whole isolation of this split process performs a and y, while its whole coisolation performs b, x, and z. Both the isolation and the coisolation perform also a plethora of auxiliary actions, represented by α, β, $\overline{\alpha}$, and $\overline{\beta}$, used to appropriately synchronize those processes. It requires only a few extra steps to show that this split process is provably equal to $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}$, which in turn is provably equal to $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle$, but as this is a result that Theorem 3 already gave us, we skip those steps here.

To implement the two parallel processes above, we must implement all auxiliary actions in α, β, $\overline{\alpha}$, and $\overline{\beta}$. Not only is this inconvenient, it is actually unnecessary (in the sense that ? can already bring about appropriate synchronization with fewer of those actions). To show this, we proceed with the second way in which we can split along $\{\mathtt{a}, \mathtt{y}\}$, namely by splitting $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}$ (instead of splitting $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle$ as done above) with respect to $\mathbb{A} = \{\mathtt{a}, \mathtt{y}\}$. This yields the following:

$$\begin{aligned}& \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\dagger}\\ \simeq\ & \mathsf{split}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d) \cdot \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow})\\ =\ & \mathsf{split}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d)) \cdot \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\dagger}\end{aligned}$$

As before, we introduce the following process definitions:

$$\begin{aligned}\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\ddagger} &= ?(\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S} \parallel \overline{\mathtt{Sequencer2}}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S})\\ \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S} &= \mathsf{isol}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d)) \cdot \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S}\\ \overline{\mathtt{Sequencer2}}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S} &= \overline{\mathsf{isol}}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d)) \cdot \overline{\mathtt{Sequencer2}}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S}\end{aligned}$$

Using RSP, one can prove $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\dagger} \simeq \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\ddagger}$ (see Footnote 13). Applying that definition yields the following:

$$\simeq\ \mathsf{split}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d)) \cdot ?(\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S} \parallel \overline{\mathtt{Sequencer2}}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S})$$

Again as before, we use [28, Appendix D, Lemma 6, page 87], to justify the following steps:

$$\begin{aligned}\simeq\ & ?(\quad (\mathsf{isol}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d)) \cdot \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S})\\ & \parallel (\overline{\mathsf{isol}}(\epsilon, \mathtt{a}(d) \sqcup \mathtt{y}(d) \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d)) \cdot \overline{\mathtt{Sequencer2}}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S}))\\ =\ & ?(\quad (\mathtt{a}(d) \sqcup \mathtt{y}(d) \sqcup \alpha \cdot \qquad\qquad \overline{\beta} \cdot \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S})\\ & \parallel (\qquad\quad \overline{\alpha} \cdot \mathtt{b}(d) \sqcup \mathtt{x}(d) \sqcup \mathtt{z}(d) \sqcup \beta \cdot \overline{\mathtt{Sequencer2}}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}^{\S}))\end{aligned}$$

For:

$$\begin{aligned}\alpha &= \xi_1(\mathtt{a})(d) \sqcup \xi_1(\mathtt{y})(d) \qquad \beta = \overline{\xi}_2(\mathtt{b})(d) \sqcup \overline{\xi}_2(\mathtt{x})(d) \sqcup \overline{\xi}_2(\mathtt{z})(d)\\ \overline{\alpha} &= \overline{\xi}_1(\mathtt{a})(d) \sqcup \overline{\xi}_1(\mathtt{y})(d) \qquad \overline{\beta} = \xi_2(\mathtt{b})(d) \sqcup \xi_2(\mathtt{x})(d) \sqcup \xi_2(\mathtt{z})(d)\end{aligned}$$

As with splitting $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle$, the whole isolation of this split process performs a and y, while its whole coisolation performs b, x, and z (this is not surprising because $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle \simeq \mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}$ and split is semantics-preserving). However, splitting $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}$ introduces fewer auxiliary actions (because the number of actions in $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}$ is less than the number of actions in $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle$). From an implementation point of view, we therefore prefer splitting $\mathtt{Sequencer2}\langle ; \mathtt{a}, \mathtt{b}\rangle_{\downarrow}$. Generally, when applying the global approach, one should first simplify a parallel process to a sequential process and afterward split that sequential process.

Local approach In the local approach, the aim is to keep as much parallel processes as possible and split only some of those based on their local (a)synchronous regions. Afterward, we can implement and run each of those parallel processes, including the "local" isolations and the "local" coisolations, in parallel. This local splitting approach corresponds to the distributed implementation approach to Reo (see Section 1).

First, we determine for which of the parallel components of the process referred to by $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ splitting makes sense (i.e., which of those parallel components consists of two synchronous regions connected by an asynchronous region), namely for $\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle$ and $\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle$. Splitting those processes yields $\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\dagger$ and $\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\dagger$. As before, we introduce the following process definitions:

$$\begin{aligned}
\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\ddagger &= ?(\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S \parallel \overline{\texttt{Fifo1}}\langle\texttt{x}_2;\texttt{y}_1\rangle^\S)\\
\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S &= \mathsf{isol}(\epsilon\,,\,\texttt{y}_1(d)\cdot\texttt{x}_2(d))\cdot\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S\\
\overline{\texttt{Fifo1}}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S &= \overline{\mathsf{isol}}(\epsilon\,,\,\texttt{y}_1(d)\cdot\texttt{x}_2(d))\cdot\overline{\texttt{Fifo1}}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S
\end{aligned}$$

$$\begin{aligned}
\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\ddagger &= ?(\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S \parallel \overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S)\\
\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S &= \mathsf{isol}(\epsilon\,,\,\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\texttt{y}_3(x)\cdot\texttt{z}_1(x)))\cdot\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S\\
\overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S &= \overline{\mathsf{isol}}(\epsilon\,,\,\textstyle\sum_{x\in\mathbb{D}\mathsf{ata}}(\texttt{y}_3(x)\cdot\texttt{z}_1(x)))\cdot\overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S
\end{aligned}$$

Using RSP, one can prove $\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\dagger \simeq \texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_2\rangle^\ddagger$ and $\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\dagger \simeq \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\ddagger$ (see Footnote 13). Replacing $\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle$ and $\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle$ with $\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\ddagger$ and $\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\ddagger$ in the process referenced by $\texttt{Sequencer2}\langle;\texttt{a},\texttt{b}\rangle$ then yields the following provably equal process:

$$\partial_\Gamma\left(\begin{array}{ccccc}
 & & \texttt{B}\langle\widetilde{\texttt{a}}_1\rangle & \parallel & \\
 & & \texttt{Sync}\langle\texttt{y}_2;\texttt{a}_1\rangle & \parallel & \\
?\left(\begin{array}{c}\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S\\ \overline{\texttt{Fifo1}}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S\end{array}\right) & \parallel & \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle & \parallel & ?\left(\begin{array}{c}\texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S\\ \overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S\end{array}\right)\\
\langle\widetilde{\texttt{x}}_1;\widetilde{\texttt{x}}_2\rangle & \parallel & & & \texttt{Sync}\langle\texttt{z}_3;\texttt{x}_1\rangle
\end{array}\right.$$
$$\left.\begin{array}{c}\texttt{B}\langle\widetilde{\texttt{b}}_1\rangle \parallel\\ \texttt{Sync}\langle\texttt{z}_2;\texttt{b}_1\rangle \parallel\\ \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle \parallel\end{array}\right)$$

By the same reasoning as before, we can push ? to the top, yielding the following provably equal process:

$$?\left(\partial_\Gamma\left(\begin{array}{ccccccc}
 & & \texttt{B}\langle\widetilde{\texttt{a}}_1\rangle & \parallel & & & \\
 & & \texttt{Sync}\langle\texttt{y}_2;\texttt{a}_1\rangle & \parallel & & & \\
\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S & \parallel & \overline{\texttt{Fifo1}}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S & \parallel & \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle & \parallel & \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S\\
\langle\widetilde{\texttt{x}}_1;\widetilde{\texttt{x}}_2\rangle & \parallel & & & & & \\
 & & & & & & \\
 & & & & & & \\
\parallel & & \overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S & \parallel & \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle & \parallel & \\
 & & & & \texttt{Sync}\langle\texttt{z}_3;\texttt{x}_1\rangle & &
\end{array}\right)\right)$$

We can now implement each of those processes and run them in parallel while ensuring two kinds of synchronization: the kind of synchronization represented by ∂_Γ (imposed by the semantics of Reo for synchronizing channel/node ends) and the kind of synchronization represented by ? (imposed by our splitting procedure for synchronizing auxiliary actions). To avoid the former kind of synchronization at run-time, one can optionally compute it "off-line" at compile-time (to some extent shifting from the distributed to the centralized approach) and implement the following parallel composition of only two processes:

$$\begin{array}{lll}
?(& (\texttt{a}(d)\sqcup\texttt{y}(d)\sqcup\xi_1(\texttt{y}_1)(d)\sqcup\xi_1(\texttt{y}_3)(d)\cdot & \bar{\xi}_2(\texttt{z}_1)(d)\sqcup\bar{\xi}_2(\texttt{x}_2)(d)\cdot\partial_\Gamma(p))\\
\parallel (& \bar{\xi}_1(\texttt{y}_1)(d)\sqcup\bar{\xi}_1(\texttt{y}_3)(d)\cdot\texttt{b}(d)\sqcup\texttt{x}(d)\sqcup\texttt{z}(d)\sqcup\xi_2(\texttt{z}_1)(d)\sqcup\xi_2(\texttt{x}_2)(d)\cdot\partial_\Gamma(q))) &
\end{array}$$

For:

$$p=\left(\begin{array}{ccccccc}
 & & \texttt{B}\langle\widetilde{\texttt{a}}_1\rangle & \parallel & & & \\
 & & \texttt{Sync}\langle\texttt{y}_2;\texttt{a}_1\rangle & \parallel & & & \\
\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S & \parallel & \overline{\texttt{Fifo1}}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S & \parallel & \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle & \parallel & \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S\\
\langle\widetilde{\texttt{x}}_1;\widetilde{\texttt{x}}_2\rangle & \parallel & & & & & \\
 & & & & & & \\
 & & & & & & \\
\parallel & & \overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S & \parallel & \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle & \parallel & \\
 & & & & \texttt{Sync}\langle\texttt{z}_3;\texttt{x}_1\rangle & &
\end{array}\right.$$
$$\left.\begin{array}{c}\texttt{B}\langle\widetilde{\texttt{b}}_1\rangle \parallel\\ \texttt{Sync}\langle\texttt{z}_2;\texttt{b}_1\rangle \parallel\\ \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle \parallel\end{array}\right)$$

And:

$$q=\left(\begin{array}{ccccccc}
 & & \texttt{B}\langle\widetilde{\texttt{a}}_1\rangle & \parallel & & & \\
 & & \texttt{Sync}\langle\texttt{y}_2;\texttt{a}_1\rangle & \parallel & & & \\
\texttt{Fifo1}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S & \parallel & \overline{\texttt{Fifo1}}_d\langle\texttt{x}_2;\texttt{y}_1\rangle^\S & \parallel & \texttt{R}\langle\widetilde{\texttt{y}}_1;\widetilde{\texttt{y}}_2,\widetilde{\texttt{y}}_3\rangle & \parallel & \texttt{Fifo1}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S\\
\langle\widetilde{\texttt{x}}_1;\widetilde{\texttt{x}}_2\rangle & \parallel & & & & & \\
 & & & & & & \\
 & & & & & & \\
\parallel & & \overline{\texttt{Fifo1}}\langle\texttt{y}_3;\texttt{z}_1\rangle^\S & \parallel & \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle & \parallel & \\
 & & & & \texttt{Sync}\langle\texttt{z}_3;\texttt{x}_1\rangle & &
\end{array}\right.$$
$$\left.\begin{array}{c}\texttt{B}\langle\widetilde{\texttt{b}}_1\rangle \parallel\\ \texttt{Sync}\langle\texttt{z}_2;\texttt{b}_1\rangle \parallel\\ \texttt{R}\langle\widetilde{\texttt{z}}_1;\widetilde{\texttt{z}}_2,\widetilde{\texttt{z}}_3\rangle \parallel\end{array}\right)$$

7. Related work

Process decomposition Closest to the process algebraic work presented in this paper seems the work on processes *decomposition*, first investigated by Milner and Moller in the late 1980s–early 1990s [40]. In that work, Milner and Moller define the notion of a *prime process*, and they explore what kind of processes p have a unique decomposition into primes $p_1,\ldots,p_k$ such that the parallel composition of those primes is strongly bisimilar to p.[23] A process p qualifies as a prime process if, for all q and r, it holds that $p\simeq q\parallel r$ implies that either q or r—not both—is equivalent to the neutral element for $\parallel$ (the

[23] The parallel composition operator differs slightly from the one in this paper: the operator used by Milner and Moller satisfies $q\parallel r\simeq q\between r+r\between q+q\mid r$ (by Axiom M in Fig. 6).

algebra used in this paper does not have such an element). In other words, one cannot decompose p further into nonneutral processes. Among other results, Milner and Moller show that finite processes in the algebra they consider have a unique prime decomposition under strong bisimulation. In his PhD thesis, Moller additionally gives a unique decomposition result with respect to (weak) observational congruence [41, Section 4.4].

After Milner and Moller, also other researchers investigated process decomposition for various process calculi. This led to some interesting applications. For instance, Lanese et al. proved a prime decomposition theorem for a higher-order process calculus and used it to prove the completeness of the axiomatization of that calculus [37]. Aceto et al. [1] and Christensen [12] used prime decomposition theorems for a similar purpose, among other contributions. Alternatively, Groote and Moller used process decomposition for verification [21]: they showed that instead of checking $p \simeq q$ directly, in some cases, one can more efficiently check whether the primes of p and q are equivalent (while preserving soundness and completeness). The projection operator introduced by Groote and Moller for decomposing processes seems somewhat related to our functions isol and $\overline{\mathsf{isol}}$, albeit rather distantly. Applied to a process p, similar to isol and $\overline{\mathsf{isol}}$, this projection operator throws some actions from p away and keeps others for communicating with other processes. However, those preserved communication actions must already occur in both the original p and the original other processes; the projection operator does not add auxiliary actions the same way isol and $\overline{\mathsf{isol}}$ do (more significant differences between process decomposition and process splitting follow shortly).

Other contributions to the theory of process decomposition include the work of Kučera [36] (decidability results and constructions of decompositions), Luttik and van Oostrom [39] (generalization of decomposition to partial commutative monoids), Luttik [38] (unique parallel decomposition modulo branching and weak bisimilarity), and Dreier et al. [17] (decomposition in the applied π-calculus).

Although related, the work on process decomposition differs significantly from our work on process splitting. For one thing, even though both approaches derive smaller processes from an existing one (such that their parallel composition is equivalent to the original process), the notion of "smaller" in our work does not involve primality. In fact, one could argue that the processes resulting from our splitting procedure are not really smaller than the original process due to the introduction of auxiliary actions. Another difference concerns uniqueness, which plays no explicit role in our splitting procedure. Note, however, that only one isolation and only one coisolation exists for every process under some fixed $\mathbb{A}$ and Ξ (due to the deterministic definition of split). So technically, we have uniqueness. Finally, in process decomposition, one usually requires no additional synchronization on top of the parallel composition of the primes. We, in contrast, needed to introduce the ? operator to achieve appropriate synchronization between the isolation and the coisolation of a process.

Connector decomposition In this paper, we developed a process algebraic splitting procedure, which we then applied to Reo's process algebraic semantics, thereby effectively splitting connectors. Interestingly, different notions of splitting and decomposition of Reo connectors—or their semantics—already exist in the literature. Although inapplicable for our purpose, we discuss them below.

Koehler and Clarke investigated the decomposition of *port automata* [31], an operational model of connector behavior. The states of a port automaton represent the internal configurations of a connector; its transitions, labeled with sets of firing node names, describe atomic execution steps. Through special product and hiding operators on port automata, one can compositionally construct a connector model from a set of smaller automata for the primitive Reo connectors. Koehler and Clarke showed that they can decompose every port automaton into instances of only two primitive automata. Essentially, this means that one can construct every Reo connector expressible by a port automaton from instances of only two different primitive connectors.

Pourvatan et al. explored the decomposition of *complete constraint automata* [42], a more expressive operational model of connector behavior than port automata and an extension of ordinary constraint automata [5]. Their approach differs significantly from the work of Koehler and Clarke: Pourvatan et al. develop a notion of inverse for their automata, which allows them to factor out certain parts of a complete constraint automaton based on another such automaton. A typical application of this decomposition technique is connector synthesis. Suppose that we have a specification (as an automaton) of the whole system that we want to build and specifications (also as automata) of the components that this system consists of, but no specification of the connector that should connect those components. We can then factor out the component automata from the system automaton to get the automaton specifying the behavior of the connector. Pourvatan et al. exemplify this with a service-oriented application.

Although not often considered (exceptions exist though—see, e.g., [13]), we remark that Arbab mentioned a split operation already in his introductory paper on Reo [2]. However, this split operation splits nodes instead of connectors (i.e., sets of nodes). Because our interest lies in splitting connectors, we could not use Arbab's notion of splitting.

Proença pioneered the work on (a)synchronous regions, region-based optimization techniques for Reo implementations, and connector splitting in this PhD thesis and associated publications [43–45]. He developed the first working Reo implementation based on these ideas, demonstrated its merits through benchmarks, and invented a new automaton model—*behavioral automata* [45]—to reason about split connectors. Also, Proença formulated a number of soundness and completeness criteria for when a split behavioral automaton preserves the semantics of the original (but without proofs). Recently, Clarke and Proença explored connector splitting in the context of the connector coloring semantics [15]. They discovered that the standard version of that semantics has undesirable properties in the context of splitting: some split connectors that intuitively *should* be equivalent to the original connector are not equivalent under the standard model. To

address this problem, Clarke and Proença propose a new variant called *partial connector coloring*, which allows one to better model locality and independences between different parts of a connector.

8. Conclusion and future work

We presented a procedure for splitting processes in a process algebra with multiactions and data (the untimed subset of the specification language mCRL2). This splitting procedure cuts a process into two processes along a set of actions $\mathbb{A}$: roughly, one of these processes contains no actions from $\mathbb{A}$, while the other process contains only actions from $\mathbb{A}$. We stated and proved a theorem asserting that the parallel composition of these two processes is provably equal from a set of axioms (sound and complete with respect to strong bisimilarity) to the original process under some appropriate notion of synchronization.

We applied our splitting procedure to the process algebraic semantics of the coordination language Reo: using this procedure and its related theorem, we formally established the soundness of splitting Reo connectors along the boundaries of their (a)synchronous regions in implementations of Reo. Such splitting can significantly improve the performance of connectors as shown elsewhere [15,43,44].

Our work shows the feasibility of using the language mCRL2 (not the associated toolset) for proving properties of a whole language, Reo, rather than of concrete connectors. This subtly, yet significantly, differs from the work presented in [35, 32–34]. In those paper, Kokash et al. introduce the process algebraic semantics of Reo for verifying concrete connectors (e.g., "this connector never deadlocks") but obtain no results about Reo as a language.

Although inspired by Reo, our splitting procedure may be useful also in other contexts. For instance, a possible application beside Reo is *projection* in choreography languages [9,10,18,19,11,23]. A projection maps a global protocol specification among k parties, called *choreography*, to k local specifications of per-party observable behavior, called *contracts* [9,10] (or *peers* [18,19] or *end-point processes* [11,23]). The challenge is to project such that the collective behavior of the resulting contracts *conforms* with the projected choreography. Interestingly, for some choreographies, without adding extra communication actions to their original specification, no projection to contracts exists that satisfies the conformance requirement. The theory presented in this paper may constitute a step in a projection method that alleviates this problem by automatically inferring which communication actions need be added to otherwise unprojectable choreographies. Roughly, the idea is to represent a choreography as an mCRL2 process (in fact, many choreography models are based on process algebra). However, rather than directly projecting this *choreography process* as customary, we propose to first decompose it into small, parallel components using an adapted version (to mCRL2) of the algorithm in [4]. The purpose of this decomposition is revealing the previously "hidden" communication actions necessary to make the original choreography projectable. We subsequently compute a *contract process* for each of the k parties in the choreography by (i) splitting some of the small, parallel components using our splitting procedure and (ii) afterward simplifying the parallel components—including the split ones—back to k sequential processes, separated from each other by split processes. By the correctness of our splitting procedure, the appropriately synchronized parallel composition of the k resulting contract processes is provably equal to the original choreography process (i.e., the conformance requirement holds). More generally, our splitting procedure may play a role in proving the correctness of parallelization techniques for sequential processes.

We identify several directions for future work.

- Implementing the splitting procedure to facilitate automatic splitting of processes, as well as a tool for the automatic detection of (a)synchronous regions of Reo connectors. Combined, they allow us to mechanically split connectors along their (a)synchronous regions. We can then integrate this in one of the code generation frameworks currently under development for Reo.
- Mechanizing our proofs using a theorem prover. We proved all propositions, lemmas, and theorems presented in this paper by hand, which was a laborious task, even though many of the equational derivations required limited creativity. As such, these proofs lend themselves to automation. Doing so has several advantages: it can confirm that our proofs are indeed correct and it may make extending those proofs to different process algebras (see next item) and/or splitting procedures (see last item) easier.
- Extending our splitting procedure to full mCRL2, including time. Time is modeled using data values from the sort $\mathbb{R}^{\geqslant 0}$ (i.e., the nonnegative reals). This sort contains infinitely many elements over which summation can be defined. In fact, one of the basic time axioms of full mCRL2—Axiom T3—involves this kind of infinite summation [20]:

  $$p \simeq \sum_{x \in \mathbb{R}^{\geqslant 0}} p \,@\, x \;\; \textbf{if} \;\; x \notin \mathsf{Free}(p)$$

  Informally, this axiom states that every (untimed) process p is provably equal to the same (but timed) process whose first multiaction or deadlock can occur at any time point (the $@$ operator is the *at* operator: $p\,@\,x$ means that the first multiaction or deadlock of p happens at time x). Thus, the first problem we need to solve is extending our current theory from finite to infinite summation. Of course, we can avoid this problem by considering only processes that terminate in finite time. Although that may be a useful first exercise (actually, we do not anticipate major issues there: we expect our main technique of adding auxiliary actions for tracking choices to work without fundamental changes), our goal is to support infinite executions (as in this paper). Extending summation from finite to infinite domains, however, seems nontrivial (see Footnote 17).

- Investigating other ways of splitting processes, corresponding to other versions of the region-based optimization technique (see Section 1). The procedure we introduced in this paper splits processes in a synchronous manner such that $\xi(a)$ occurs at the same time as the action a itself. We imagine at least two other ways of splitting processes. In one approach, $\xi(a)$ occurs after a but before the next action. Then, the process $q = a \cdot b$ has $a \cdot \xi(a) \cdot \overline{\xi}(b)$ as its $\{a\}$-isolation (instead of $a \sqcup \xi(a) \cdot \overline{\xi}(b)$). In another approach, $\xi(a)$ occurs after a but possibly concurrently with the next action. Then, q has $a \cdot (\xi(a) \parallel \overline{\xi}(b))$ as its isolation. We speculate that these splitting approaches are sound only under equivalences weaker than strong bisimulation.

  This particular line of future work seems related to existing work on delay-insensitive circuits (e.g., [46]) and desynchronization (e.g., [6,16]), the derivation of an asynchronous system from a synchronous system: for the class of *desynchronizable systems*, the original synchronous system and the newly constructed asynchronous system are equivalent. If we use the splitting procedure presented in this paper to obtain such an original synchronous system, we may use—perhaps with modifications—results from desynchronization for more asynchronous splitting.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.scico.2014.02.017.

References

[1] Luca Aceto, Wan Fokkink, Anna Ingolfsdottir, Bas Luttik, Split-2 bisimilarity has a finite axiomatization over CCS with Hennessy's merge, Log. Methods Comput. Sci. 1 (1) (2005) 1–12.
[2] Farhad Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.
[3] Farhad Arbab, Puff, the magic protocol, in: Gul Agha, Olivier Danvy, José Meseguer (Eds.), Talcott Festschrift, in: LNCS, vol. 7000, Springer, 2011, pp. 169–206.
[4] Farhad Arbab, Christel Baier, Frank De Boer, Jan Rutten, Marjan Sirjani, Synthesis of Reo circuits for implementation of component-connector automata specifications, in: Jean-Marie Jacquet, Gian Pietro Picco (Eds.), Coordination Models and Languages, in: LNCS, vol. 3454, Springer, 2005, pp. 236–251.
[5] Christel Baier, Marjan Sirjani, Farhad Arbab, Jan Rutten, Modeling component connectors in Reo by constraint automata, Sci. Comput. Program. 61 (2) (2006) 75–113.
[6] Harsh Beohar, Pieter Cuijpers, A theory of desynchronisable closed loop systems, in: Simon Bliudze, Roberto Bruni, Davide Grohmann, Alexandra Silva (Eds.), Proceedings ICE 2010, in: EPTCS, vol. 38, CoRR, 2010, pp. 99–114.
[7] Jan Bergstra, Jan Willem Klop, Process algebra for synchronous communication, Inf. Control 60 (1–3) (1984) 109–137.
[8] Jan Bergstra, Jan Willem Klop, Verification of an alternating bit protocol by means of process algebra protocol, in: Wolfgang Bibel, Klaus Jantke (Eds.), Mathematical Methods of Specification and Synthesis of Software Systems '85, in: LNCS, vol. 215, Springer, 1986, pp. 9–23.
[9] Mario Bravetti, Gianluigi Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: Markus Lumpe, Wim Vanderperren (Eds.), Software Composition, in: LNCS, vol. 4829, Springer, 2007, pp. 34–50.
[10] Mario Bravetti, Gianluigi Zavattaro, Contract compliance and choreography conformance in the presence of message queues, in: Roberto Bruni, Karsten Wolf (Eds.), Web Services and Formal Methods, in: LNCS, vol. 5387, Springer, 2009, pp. 37–45.
[11] Marco Carbone, Kohei Honda, Nobuko Yoshida, Structured communication-centered programming for web services, ACM Trans. Program. Lang. Syst. 34 (2) (2012) 8:1–8:78.
[12] Søren Christensen, Decidability and decomposition in process algebras, PhD thesis, University of Edinburgh, 1993.
[13] Dave Clarke, A basic logic for reasoning about connector reconfiguration, Fundam. Inform. 82 (4) (2008) 361–390.
[14] Dave Clarke, David Costa, Farhad Arbab, Connector colouring I: Synchronisation and context dependency, Sci. Comput. Program. 66 (3) (2007) 205–225.
[15] Dave Clarke, José Proença, Partial connector colouring, in: Marjan Sirjani (Ed.), Coordination Models and Languages, in: LNCS, vol. 7274, Springer, 2012, pp. 59–73.
[16] Clemens Fischer, Wil Janssen, Synchronous development of asynchronous systems, in: Ugo Montanari, Vladimiro Sassone (Eds.), CONCUR '96: Concurrency Theory, in: LNCS, vol. 1119, Springer, 1996, pp. 735–750.
[17] Jannik Dreier, Cristian Ene, Pascal Lafourcade, Yassine Lakhnech, On unique decomposition of processes in the applied π-calculus, in: Frank Pfenning (Ed.), Foundations of Software Science and Computation Structures, in: LNCS, vol. 7794, Springer, 2013, pp. 50–64.
[18] Xiang Fu, Tevfik Bultan, Jianwen Su, Conversation protocols: a formalism for specification and verification of reactive electronic services, Theor. Comput. Sci. 328 (1–2) (2004) 19–37.
[19] Xiang Fu, Tevfik Bultan, Jianwen Su, Realizability of conversation protocols with message contents, Int. J. Web Serv. Res. 2 (4) (2005) 68–93.
[20] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, Muck van Weerdenburg, The formal specification language mCRL2, in: Proceedings of MMOSS, 2007, pp. 1–34.
[21] Jan Friso Groote, Faron Moller, Verification of parallel systems via decomposition, in: Walter Cleaveland (Ed.), CONCUR '92, in: LNCS, vol. 630, Springer, 1992, pp. 62–76.
[22] Jan Friso Groote, Mohammad Reza Mousavi, Modelling and Analysis of Communicating Systems, available at http://www.win.tue.nl/~jfg/educ/2IW26/herfst2011/mcrl2-book.pdf, 2010.
[23] Kohei Honda, Nobuko Yoshida, Marco Carbone, Multiparty asynchronous session types, in: Proceedings of POPL, ACM, 2008, pp. 273–284.
[24] Sung-Shik Jongmans, Farhad Arbab, Overview of thirty semantic formalisms for Reo, Sci. Ann. Comput. Sci 22 (1) (2012) 201–251.
[25] Sung-Shik Jongmans, Farhad Arbab, Global consensus through local synchronization, in: Natallia Kokash, Javier Cámara (Eds.), Proceedings of FOCLASA, 2013.
[26] Sung-Shik Jongmans, Farhad Arbab, Modularizing and specifying protocols among threads, in: Simon Gay, Paul Kelly (Eds.), Proceedings of PLACES 2012, in: EPTCS, vol. 109, CoRR, 2013, pp. 34–45.
[27] Sung-Shik Jongmans, Dave Clarke, José Proença, A procedure for splitting processes and its application to coordination, in: Natallia Kokash, António Ravara (Eds.), Proceedings of FOCLASA 2012, in: EPTCS, vol. 91, CoRR, 2012, pp. 79–96.
[28] Sung-Shik Jongmans, Dave Clarke, José Proença, A procedure for splitting data-aware processes and its application to coordination, Technical Report, CWI, FM-1401, 2014, http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-22358.
[29] Sung-Shik Jongmans, Christian Krause, Farhad Arbab, Encoding context-sensitivity in Reo into non-context-sensitive semantic models, in: Gruia-Catalin Roman, Wolfgang de Meuter (Eds.), Coordination Models and Languages, in: LNCS, vol. 6721, Springer, 2011, pp. 31–48.

[30] Sung-Shik Jongmans, Francesco Santini, Mahdi Sargolzaei, Farhad Arbab, Hamideh Afsarmanesh, Automatic code generation for the orchestration of web services with Reo, in: Flavio de Paoli, Ernesto Pimentel, Gianluigi Zavattaro (Eds.), Service-Oriented and Cloud Computing, in: LNCS, vol. 7592, Springer, 2012, pp. 1–16.
[31] Christian Koehler, Dave Clarke, Decomposing port automata, in: Michael Schumacher, Alan Wood (Eds.), Proceedings of SAC, ACM, 2009, pp. 1369–1373.
[32] Natallia Kokash, Christian Krause, Erik de Vink, Data-aware design and verification of service compositions with Reo and mCRL2, in: Manuel Mazzara, Claudio Guidi, Ivan Lanese (Eds.), Proceedings of SAC, ACM, 2010, pp. 2406–2413.
[33] Natallia Kokash, Christian Krause, Erik de Vink, Time and data-aware analysis of graphical service models in Reo, in: José Luiz Fiadeiro, Stefania Gnesi (Eds.), Proceedings of SEFM, IEEE, 2010, pp. 125–134.
[34] Natallia Kokash, Christian Krause, Erik de Vink, Verification of context-dependent channel-based service models, in: Frank de Boer, Marcello Bonsangue, Stefan Hallerstede, Michael Leuschel (Eds.), Formal Methods for Components and Objects, in: LNCS, vol. 6286, Springer, 2010, pp. 21–40.
[35] Natallia Kokash, Christian Krause, Erik de Vink, Reo+mCRL2: A framework for model-checking dataflow in service compositions, Form. Asp. Comput. 24 (2) (2012) 187–216.
[36] Antonín Kučera, Effective decomposability of sequential behaviours, Theor. Comput. Sci. 242 (1–2) (2000) 71–89.
[37] Ivan Lanese, Jorge Pérez, Davide Sangiorgi, Alan Schmitt, On the expressiveness and decidability of higher-order process calculi, Inf. Comput. 209 (2) (2011) 198–226.
[38] Bas Luttik, Unique parallel decomposition in branching and weak bisimulation semantics, in: Jos Baeten, Tom Ball, Frank de Boer (Eds.), Theoretical Computer Science, in: LNCS, vol. 7604, Springer, 2012, pp. 250–264.
[39] Bas Luttik, Vincent van Oostrom, Decomposition orders—another generalisation of the fundamental theorem of arithmetic, Theor. Comput. Sci. 335 (2–3) (2005) 147–186.
[40] Robin Milner, Faron Moller, Unique decomposition of processes, Theor. Comput. Sci. 107 (2) (1993) 357–363.
[41] Faron Moller, Axioms for concurrency, PhD thesis, University of Edinburgh, 1989.
[42] Bahman Pourvatan, Marjan Sirjani, Farhad Arbab, Marcello Bonsangue, Decomposition of constraint automata, in: Luís Barbosa, Markus Lumpe (Eds.), Formal Aspects of Component Software, in: LNCS, vol. 6921, Springer, 2012, pp. 237–258.
[43] José Proença, Dave Clarke, Erik de Vink, Farhad Arbab, Dreams: a framework for distributed synchronous coordination, in: Mirko Viroli, Gabriella Castelli, Jose Luis Fernandez Marquez (Eds.), Proceedings of SAC, ACM, 2012, pp. 1510–1515.
[44] José Proença, Synchronous coordination of distributed components, PhD thesis, Leiden University, 2011.
[45] José Proença, Dave Clarke, Erik de Vink, Farhad Arbab, Decoupled execution of synchronous coordination models via behavioural automata, in: Mohammad-Reza Mousavi, António Ravara (Eds.), Proceedings of FOCLASA 2011, in: EPTCS, vol. 58, CoRR, 2011, pp. 65–79.
[46] Jan Tijmen Udding, Classification and composition of delay-insensitive circuits, PhD thesis, Eindhoven University of Technology, 1984.
[47] Yaroslav Usenko, Linearization in μCRL, PhD thesis, Eindhoven University of Technology, 2004.