Science of Computer Programming 115–116 (2016) 199–224

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Global consensus through local synchronization: A formal basis for partially-distributed coordination

S.-S.T.Q. Jongmans [a,b,*], F. Arbab [b,a]

[a] *Universiteit Leiden, Niels Bohrweg 1, 2333 CA, Leiden, Netherlands*
[b] *Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, Netherlands*

A R T I C L E   I N F O

Article history:
Received 9 March 2014
Received in revised form 7 August 2015
Accepted 14 September 2015
Available online 24 September 2015

Keywords:
Reo
Port automata
Nonassociative product
Parallel composition
Connector implementation

A B S T R A C T

A promising new application domain for coordination languages is expressing interaction protocols among threads/processes in multicore programs: coordination languages typically provide high-level constructs and abstractions that more easily compose into correct (with respect to a programmer's intentions) protocol specifications than do low-level synchronization constructs (e.g., locks, semaphores, etc.) provided by conventional languages. However, a crucial step toward adoption of coordination languages for multicore programming is the development of compiler technology: programmers must have tools to automatically generate efficient code for high-level protocol specifications.

In ongoing work, we are developing compilers for a coordination language, Reo, based on that language's automata semantics. As part of the compilation process, our tool computes the product of a number of automata, each of which models a constituent of the protocol to generate code for. This approach ensures that implementations of those automata at runtime reach a consensus about their global behavior in every step. However, this approach has two problems: state space explosion at compile-time and oversequentialization at runtime. In this paper, we provide a solution by defining a new, local product operator on those automata that avoids these problems. We then identify a sufficiently large class of automata for which using our new product instead of the existing one is semantics-preserving.

1. Introduction

Context Coordination languages have emerged for the specification and implementation of interaction protocols among concurrent processes (services, threads, etc.). This class of languages includes Reo [1,2], a graphical language for compositional construction of protocols, manifested as *connectors*: communication media through which processes can interact with each other. Fig. 1 shows example connectors in their usual graphical syntax. Briefly, connectors consist of one or more *channels*, through which data items flow, and a number of *nodes*, on which channel ends coincide. Through connector *composition* (the act of gluing connectors together on their shared nodes), users can construct arbitrarily complex connectors. To communicate data and synchronize their execution, concurrent processes perform I/O-operations on the *boundary nodes* of a connector (shown in Fig. 1 as open circles).

* Corresponding author at: Centrum Wiskunde & Informatica, Science Park 123, 1098 XG, Amsterdam, Netherlands.
E-mail addresses: jongmans@cwi.nl (S.-S.T.Q. Jongmans), farhad@cwi.nl (F. Arbab).

http://dx.doi.org/10.1016/j.scico.2015.09.001
0167-6423/© 2015 Elsevier B.V. All rights reserved.

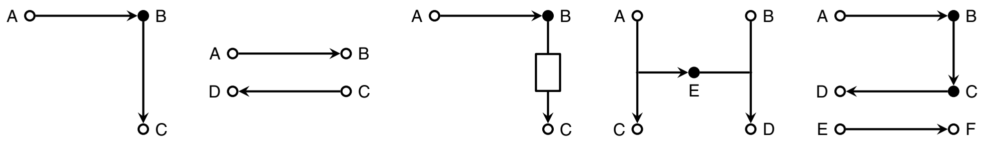

Fig. 1. Five example connectors. Open circles represent boundary nodes, on which processes perform I/O-operations; filled circles represent nodes for internal routing. The first four connectors in this figure consist of two *primitives* (i.e., minimal subconnectors); the fifth connector consists of four primitives. The two connected primitives in the first, third, and fourth connector have one shared node; the three connected primitives in the fifth connector have two shared nodes.

Reo has successfully been used to confront the problem of Web service composition, both theoretically [3,4] and practically [5,6]. Perhaps even more promising and exciting, however, is recent work on using Reo for programming multicore applications. When it comes to multicore programming, Reo has a number of advantages over conventional programming languages, which feature a fixed set of low-level synchronization constructs (locks, mutexes, etc.). Programmers using such a conventional language have to translate the synchronization needs of their protocols into the synchronization constructs of that language. Because this translation occurs in the mind of the programmer, invariably some context information either gets irretrievably lost or becomes implicit and difficult to extract in the resulting code. In contrast, Reo allows programmers to compose the protocols of their application (i.e., connectors) at a high abstraction level. Not only does this reduce the conceptual gap for programmers, which makes it easier to implement and reason about protocols, but by preserving all relevant context information, high-level protocol specifications also offer considerable novel opportunities for compilers to do optimizations on multicore hardware.

Additionally, Reo has several software engineering advantages as a domain-specific language for protocols [7]. For instance, Reo forces developers to separate their computation code from their protocol code. Such a separation facilitates verbatim reuse, independent modification, and compositional construction of protocol implementations (i.e., connectors) in a straightforward way. Moreover, Reo has a strong mathematical foundation [8], which enables formal connector analyses (e.g., model checking [9]). This makes statically verifying deadlock-freedom in a given protocol, for instance, relatively easy; such analyses are much harder to perform on code in lower-level general-purpose languages.

To use connectors for implementing protocols among concurrent processes in real applications, one must derive implementations from their graphical specification, as precompiled executable code or using a run-time interpretation engine. Roughly two implementation approaches currently exist. In the *distributed approach* [10–13], 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* [7,14,5,6], one computes the behavior of a connector as a whole, implements this behavior, and runs this implementation sequentially as a centralized system.

In ongoing work, we are developing Reo compilers (Reo-to-Java [7], Reo-to-C [14]) according to the centralized approach based on Reo's formal automata semantics [15]. On input of a graphical connector specification (as an XML file), these tools automatically generate code in four steps. *First*, they extract from the specification a list of the channels and nodes constituting the specified connector. *Second*, they consult a database to find for every channel and node in the list a "small" automaton that formally describes the behavior of that particular channel. *Third*, they compute the product of the automata in the constructed collection to obtain one "big" automaton that describes the behavior of the whole connector. *Fourth*, they feed a data structure representing that big automaton to a template. Essentially, this template is an incomplete fragment of sequential code with "holes" that need be "filled" (with information from the data structure). At run-time, the generated code simulates the big automaton computed in the third step: it runs sequentially in its own *protocol process* and responds to events (i.e., I/O operations) coming from the *computation processes* under coordination that perform the real work (i.e., its environment).

Problem Computing one big automaton (the third step of the centralized approach) and afterward translating it to sequential code (the fourth step) has two problems. First, computing the product may require too many resources at compile-time, because such computations require, in the worst case, time and space exponential in the (state space) size of the largest small automaton. Second, even if our compiler succeeds in computing the big automaton, the subsequently generated *sequential* code may unnecessarily restrict parallelism among independent transitions at run-time[1]: such sequential code serializes independent transitions, forcing them to execute one after the other (see Section 2.2 for details). Consequently, although formally sound, the generated implementation may run overly sequentially (e.g., if the first transition to execute takes a long time to complete, while other transitions could have fired manifold during that time).

One approach to solving these two problems is to *not* compute one big automaton but generate code directly for each of the small automata instead, essentially moving from the centralized approach to the distributed approach: using a distributed algorithm, the implementations of the small automata apply the product operators between them at run-time instead of at compile-time. Although this approach solves the stated problem—independent transitions can execute

[1] Independent transitions cannot disable each other by firing.

simultaneously—the necessary distributed algorithm for run-time product application generally inflicts a substantial amount of overhead.

Contribution and organization This paper provides a better solution to the stated problem by offering a middle ground between centralized and distributed approaches, wherein some subsets of small automata are statically composed to comprise a distributed system of locally centralized "medium" automata. Typically, each medium automata interacts/synchronizes with few other such medium automata for its transitions, while it represents the composition of a subset of small automata that interact/synchronize with each other relatively heavily. Taking the purely distributed approach as our starting point, we define a new product operator whose application at run-time requires only a relatively simple distributed algorithm—automata need to communicate only locally (i.e., with "neighbors") instead of globally (i.e., with everybody)—while allowing independent transitions to execute simultaneously. We then characterize a class of product automata where substituting the existing product operator with our new product operator is semantics-preserving. This class includes product automata whose constituents communicate only *asynchronously* with each other, and so, the optimization technique based on the identification of synchronous and asynchronous regions of connectors can be combined with our results [11]. Finally, to compensate for the nonassociativity of our new product operator, which may make its efficient application at run-time problematic, we introduce a proof method involving the construction of "huge" automata to show that efficient implementations in fact exist.

The value of our results extends beyond being an essential contribution to code generation technology for Reo, in at least two ways. First, because we formulate our results generally in terms of automata (i.e., Reo's semantics, essentially independent of Reo), any system expressible in terms of such automata can benefit from these results (e.g., actor-based Rebeca systems [16]). Second, our proof method, in which we compare distributed algorithms by modeling them as different product operators on automata and studying those operators' properties, is not only effective and elegant but also—as far as we know—novel. It enables formal reasoning about distributed algorithms (in particular, reasoning about their equivalence) at a different level of abstraction than, for instance, the seminal work by Lynch [17].

The rest of this paper looks as follows. In Section 2, we introduce our automata model, their existing product operator, and centralized/distributed Reo implementations based on such automata. In Section 3, we introduce our new product operator. In Section 4, we investigate under which circumstance substituting the existing product operator with our new product operator is semantics-preserving. We start with a complete characterization and afterward develop a cheaper characterization with a natural interpretation in the context of Reo. In Section 5, we study the role of associativity in distributed automata-based implementations. In Section 6, we discuss related work. Section 7 concludes this paper.

Although inspired by Reo, we express our main results in a purely automata-theoretic setting. We therefore skip an introduction to Reo; interested readers may consult [1,2].

A preliminary version of this paper was presented at the 12th International Workshop on Foundations of Coordination Languages and Self Adaptive Systems (FOCLASA 2013) and included in its proceedings [18]. This paper extends that preliminary version with proof outlines, a better explanation of the relation between our results and their implications in practice, a new section on the role of associativity in our theory, including new results, and an improved overview of related work.

2. Preliminaries

2.1. Port automata

Our compilation tools are based on *constraint automata* [15], introduced by Baier et al. as a semantics for Reo. Constraint automata are a general formalism for describing systems behavior. They have been used to model not only Reo connectors but also, for instance, actor-based systems [16]. For Reo, a constraint automaton specifies *when* during execution of a connector *which* data items flow *where*. Whenever the particular data values communicated through a connector do not matter (i.e., whenever only *synchronization* matters), constraint automata can be simplified into *port automata* (PA) [19]. Because data forms an orthogonal concern to the material presented in this paper, we focus our attention on port automata.

Structurally, every PA consists of finite sets of states, transitions between states, and ports, each of which models a binary Reo node.[2] States represent the internal configurations of a connector, while transitions describe its atomic execution steps. Every transition has a label that consists of a *synchronization constraint*, a set of ports that must synchronize in the firing of a transition. Let $\mathbb{P}\text{ORT}$ and $\mathbb{S}\text{TATE}$ denote the sets of all ports and all states.

Definition 1 *(Port automata).* A port automaton is a tuple $(Q, \mathcal{P}, \longrightarrow, \imath)$ where[3]:

- $Q \subseteq \mathbb{S}\text{TATE}$; (states)
- $\mathcal{P} \subseteq \mathbb{P}\text{ORT}$; (ports)
- $\longrightarrow \subseteq Q \times \wp(\mathcal{P}) \times Q$; (transitions)

[2] Binary nodes, on which at most two channel ends coincide, can compose into arbitrary n-ary nodes [15].

[3] Let $\wp(\_)$ denote the power set operator.

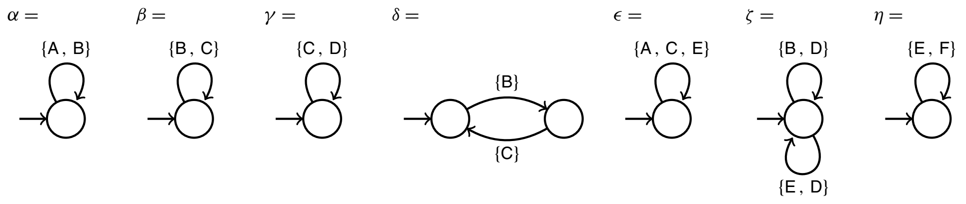

Fig. 2. Automata, denoted by α, β, γ, δ, ϵ, ζ, and η, describing the behavior of the primitives constituting the example connectors in Fig. 1: α and β model the primitives in the first connector, α and γ the primitives in the second, α and δ the primitives in the third, ϵ and ζ the primitives in the fourth, and α, β, γ, and η the primitives in the fifth.

- and $\imath \in Q$. (initial state)

$\mathbb{P}$A denotes the set of all port automata.

Fig. 2 shows example PA. For instance, the {A, B}-transition of α describes the only (infinitely repeated) execution step of the horizontal primitive, say Primitive, of the first connector in Fig. 1. In that execution step, Primitive has synchronous interaction on nodes A (a write of data d by the environment) and B (the flow of a copy of d from the horizontal to the vertical primitive). Similarly, the {A, C, E}-transition of ϵ means that the left-hand primitive of the fourth connector in Fig. 1 has synchronous interaction on nodes A (a write of data d by the environment), C (a take of a copy of d by the environment), and E (the flow of another copy of d from the left-hand to the right-hand primitive). Finally, ζ, which models the right-hand primitive of the fourth connector in Fig. 1, has two transitions. Consequently, the right-hand primitive can repeatedly choose between two steps: it has synchronous interaction either on nodes B (a write of data d by the environment) and D (a take of a copy of d by the environment) or on nodes E (the flow of data from the left-hand to the right-hand primitive) and D. However, it can fire the latter transition only if the left-hand primitive simultaneously fires its {A, C, E}-transition (otherwise, there is no data available on E to enable the firing).

If α denotes a PA, let State(α), Port(α), Trans(α), and init(α) denote its states, ports, transitions, and initial state. We adopt *strong bisimilarity* on PA as our main behavioral equivalence [19]: if α and β are bisimilar, denoted by $\alpha \approx \beta$, α can "simulate" every transition of β and vice versa. We call α and β *weakly bisimilar* if α can simulate every transition of β with a (possibly empty) sequence of *silent transitions* followed by one "normal" transition and vice versa. Silent transitions model internal execution steps of a connector and have $(\emptyset, \top)$ as their label. We use the notion of weak bisimilarity only in Section 5 and write "bisimilarity" instead of "strong bisimilarity".

Through their transitions, constraint automata model which data-flows among ports can take place. Abstracting data and their flows away, port automata merely model which set of ports engage in an atomic exchange of data on each transition. As a precondition for such a specific data-flow or synchronization of a set of ports to actually occur in practice, at run-time, every *boundary port* (i.e., a port corresponding to a boundary node of a connector) involved in that data-flow or synchronization *must* have a pending I/O-operation. In other words, every port in the synchronization constraint of a transition must be ready to participate in a firing of that transition for that transition to truly fire. Pending I/O-operations are not explicitly modeled in PA, and in that sense, PA *underspecify* connector behavior. In most cases, this is no problem. In some cases, however, the extra expressive power granted by an explicit model of pending I/O operations *is* useful, most notably for modeling connectors with *context-sensitive* behavior, where the set of admissible data-flows (i.e., the enabledness of transitions) in every instant depends on the *absence* of pending I/O-operations. In the literature, several possibilities to represent context-sensitive behavior without an explicit model of pending I/O-operations exist. One option is to *encode* information about pending I/O-operations by introducing a number of "fictitious ports" [20]. Another option is to stipulate *maximal progress* [21,22]. Maximal progress means that if multiple data-flows are possible in some instant, a connector will select one that involves as many nodes as possible. The standard semantics of Reo is more liberal: it allows a connector to nondeterministically select a data-flow among any number of possible alternatives (regardless of the number of participating nodes). As such, maximal progress strictly refines Reo's standard nondeterministic semantics. In this paper, we remain faithful to the standard nondeterministic semantics of Reo and do not stipulate maximal progress.

2.2. Product operator on PA

Individual PA describe the behavior of individual connectors; the application of the existing product operator to such PA models connector composition [19]. We define this operator in two steps. First, we introduce a relation that defines when a transition of one PA, say Alice, and a transition of another PA, say Bob, represent execution steps in which Alice and Bob *weakly agree* on their behavior. In that case, Alice and Bob agree on which of their shared ports to fire while allowing each other to simultaneously fire other ports. In the following definition, we represent a transition of Alice as a pair of port-sets: one for all Alice's ports ($\mathcal{P}_\alpha$) and one that labels a particular transition of hers (P_α). Likewise for Bob.

Definition 2 *(Weak agreement relation).* The weak agreement relation, denoted by $\Diamond$, is the relation on $\wp(\mathbb{P}\text{ORT})^2 \times \wp(\mathbb{P}\text{ORT})^2$ defined as:

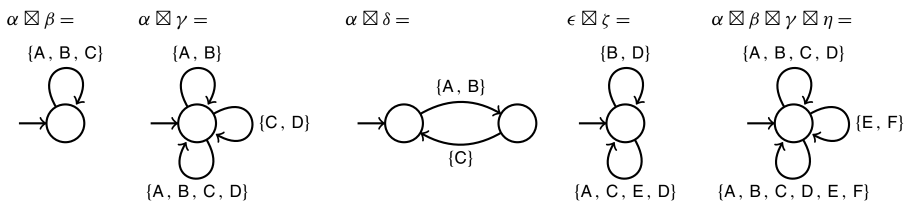
$\alpha \boxtimes \beta =$
{A, B, C}
$\alpha \boxtimes \gamma =$
{A, B}
{C, D}
{A, B, C, D}
$\alpha \boxtimes \delta =$
{A, B}
{C}
$\epsilon \boxtimes \zeta =$
{B, D}
{A, C, E, D}
$\alpha \boxtimes \beta \boxtimes \gamma \boxtimes \eta =$
{A, B, C, D}
{E, F}
{A, B, C, D, E, F}

Fig. 3. Automata describing the behavior of the example connectors in Fig. 1, constructed using $\boxtimes$ (α, β, γ, δ, ϵ, ζ, and η denote the automata in Fig. 2).

(a) Syntax	(b) Semantics
α := any element from $\mathbb{P}\text{A}$ t := α | t [x] t	$[\![\alpha]\!] = \alpha$ $[\![t \text{ [x] } t]\!] = [\![t]\!] \boxtimes [\![t]\!]$

Fig. 4. $\boxtimes$-terms.

$$(\mathcal{P}_\alpha\,,\,P_\alpha)\lozenge(\mathcal{P}_\beta\,,\,P_\beta)\ \textbf{iff}\ \left[P_\alpha\subseteq\mathcal{P}_\alpha\ \textbf{and}\ P_\beta\subseteq\mathcal{P}_\beta\ \textbf{and}\ \mathcal{P}_\alpha\cap P_\beta=\mathcal{P}_\beta\cap P_\alpha\right]$$

Next, we define the existing product operator on PA in terms of $\lozenge$.

Definition 3 *(Product operator).* The product operator, denoted by $\_\boxtimes\_$, is the operator on $\mathbb{P}\text{A}\times\mathbb{P}\text{A}$ defined by the following equation:

$$\alpha\boxtimes\beta=(\mathsf{State}(\alpha)\times\mathsf{State}(\beta)\,,\,\mathsf{Port}(\alpha)\cup\mathsf{Port}(\beta)\,,\,\longrightarrow\,,\,(\mathsf{init}(\alpha)\,,\,\mathsf{init}(\beta)))$$

where $\longrightarrow$ denotes the smallest relation induced by the following rules:

$$\frac{q_\alpha\xrightarrow{P_\alpha}_\alpha q'_\alpha\ \textbf{and}\ q_\beta\xrightarrow{P_\beta}_\beta q'_\beta\ \textbf{and}\ (\mathsf{Port}(\alpha)\,,\,P_\alpha)\lozenge(\mathsf{Port}(\beta)\,,\,P_\beta)}{(q_\alpha\,,\,q_\beta)\xrightarrow{P_\alpha\cup P_\beta}(q'_\alpha\,,\,q'_\beta)}\ (\text{WkAgr})$$

$$\frac{q_\alpha\xrightarrow{P_\alpha}_\alpha q'_\alpha\ \textbf{and}\ q_\beta\in Q_\beta\ \textbf{and}\ P_\alpha\cap\mathsf{Port}(\beta)=\emptyset}{(q_\alpha\,,\,q_\beta)\xrightarrow{P_\alpha}(q'_\alpha\,,\,q_\beta)}\ (\text{IndepA})\qquad\frac{q_\beta\xrightarrow{P_\beta}_\beta q'_\beta\ \textbf{and}\ q_\alpha\in Q_\alpha\ \textbf{and}\ P_\beta\cap\mathsf{Port}(\alpha)=\emptyset}{(q_\alpha\,,\,q_\beta)\xrightarrow{P_\beta}(q_\alpha\,,\,q'_\beta)}\ (\text{IndepB})$$

Fig. 3 shows examples of the application of $\boxtimes$. The {A, B, C, D}-transition in the second PA results from applying rule WkAgr to disjoint sets of ports. This models that two independent transitions *coincidentally* can happen simultaneously (i.e., true concurrency). The following lemma states that bisimilarity is a congruence.

Lemma 1. *(See Theorem 1, [19].)* $\left[\alpha\approx\beta\ \textbf{and}\ \gamma\approx\delta\right]\ \textbf{implies}\ \alpha\boxtimes\gamma\approx\beta\boxtimes\delta.$

Furthermore, $\boxtimes$ is associative and commutative up-to bisimilarity.

We define a $\boxtimes$-*term* as a word t in the language generated by the grammar in Fig. 4a; we define the *meaning* of $\boxtimes$-terms with the interpretation function $[\![\cdot]\!]$ in Fig. 4b.

2.3. PA-based connector implementations

Generally, a concurrent application developed with Reo consists of $n+m$ concurrent processes at run-time: $n>1$ computation processes, which perform the real computations of the application, and $m\geq 1$ protocol processes, which enforce the protocol of the application,[4] specified as a Reo connector (say, Connector) and compiled via its PA semantics. By Reo's concurrency model [1,2], computation processes have disjoint address spaces and cannot exchange messages with each other directly. Instead, every computation process has a number of formal parameters through which it can access `Port` data structures (each of which represents a port of Connector) on which that process can perform blocking I/O-operations. Computation processes share access to `Port` data structures with protocol processes when they receive the same actual parameter `Port` data structures for their respective formal parameters (essentially linking computation processes with

[4] An application may involve multiple protocols instead of just one. However, because we can consider those multiple protocols theoretically as one (consisting of disjoint "subprotocols"), we ignore this case without loss of generality.

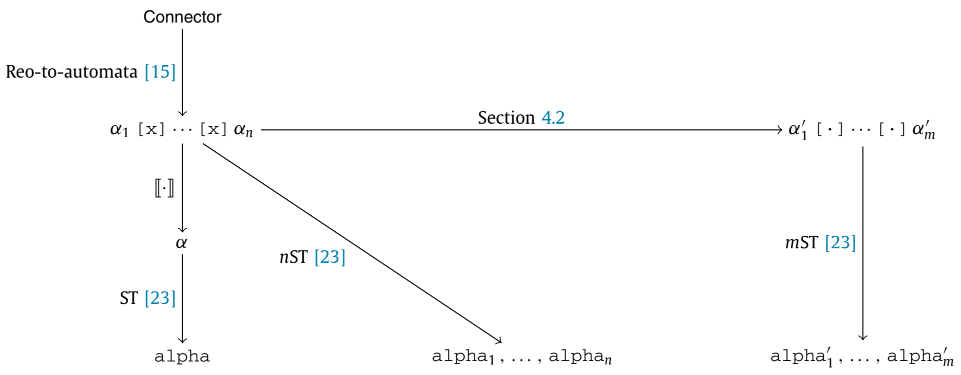

Fig. 5. Implementation approaches: centralized (left branch), distributed (middle branch), and hybrid (right branch, not discussed until Section 4.2).

Connector) such that every `Port` data structure has exactly two processes as its users: a computation process and a protocol process (possible in the centralized and distributed approach) or two protocol processes (possible only in the distributed approach). Whenever a computation process performs an I/O-operation on the `Port` data structure for a port p, this operation notifies the protocol process that has access to the same structure about this new *event* on p, or p-event. That protocol process subsequently awakes from its dormant base state and starts a new round of event-handling. In every such round, a protocol process tries to fire one transition of the PA that it should behave as, thus simulating that PA. As long as a protocol process has not yet successfully handled an event, the computation process that caused that event blocks. Next, we discuss the details of event-handling, which depends on the implementation approach.

Centralized approach The left branch in Fig. 5 shows the compilation steps involved in generating centralized implementations of Reo connectors. On input of a connector Connector, the compiler first finds a ⊠-term α_1 `[x]` … `[x]` α_n that models Connector's behavior, where atoms are the small PA for the n primitive constituents (nodes and channels) of Connector. The compiler subsequently computes the meaning of the ⊠-term (i.e., it applies product operators to PA in the term), which yields a big PA α. Finally, the compiler translates α to one piece of sequential event-handling code using ANTLR's StringTemplate template engine [23]. Consequently, a Reo-based concurrent application implemented according to the centralized approach has only one protocol process at run-time (which has access to all `Port` data structures through its formal parameters).

The protocol process has a number of internal states each of which reflects a state $q \in Q$ of the big PA $\alpha = (Q\,,\,\mathcal{P}\,,\,\longrightarrow\,,\,\imath)$. Let $B \subseteq \mathcal{P}$ denote the boundary ports of Connector. In a state q, in every event-handling round driven by a p-event, the protocol process performs the steps in Fig. 6a. We do not intend Fig. 6a to convey a real "algorithm"; it should be considered just as a stylized description of what event-handling roughly entails in the centralized approach.

Distributed approach The middle branch in Fig. 5 shows the compilation steps involved in generating distributed implementations of Reo connectors. As in the centralized approach, on input of a connector Connector, the compiler first finds a ⊠-term α_1 `[x]` … `[x]` α_n that models Connector's behavior. Contrasting the centralized approach, however, the compiler subsequently translates this ⊠-term to n pieces of sequential event-handling code, one for every small PA in the ⊠-term. Consequently, a Reo-based concurrent application implemented according to the distributed approach has as many protocol processes as Connector has constituents. Essentially, to decide which transition to fire, each of those protocol processes performs the same steps as the single protocol process in a centralized implementation. However, in distributed implementations, protocol processes must also perform a number of additional steps in which they exchange messages with each other to properly synchronize their *local transitions* (i.e., transitions in small PA), thereby forming (or collectively mimicking) one consistent global transition (i.e., a transition in their corresponding big PA, as computed in the centralized approach). These message exchanges constitute a distributed consensus algorithm that implements the product operators between the PA in the ⊠-term, thereby computing the application of those product operators at run-time. Different from centralized connector implementations, in distributed connector implementations, events originate not only from computation processes (i.e., I/O-operations on ports) but also from protocol processes. In the latter case, an event signals the arrival of a message in the mailbox of a protocol process. Protocol processes can handle both kinds of events in nearly the same way.

A protocol process for a small PA $\alpha_i = (Q_i\,,\,\mathcal{P}_i\,,\,\longrightarrow_i\,,\,\imath_i)$ has a number of internal states each of which reflects a state $q \in Q_i$. Let $B \subseteq \mathcal{P}$ denote the boundary ports of Connector. In a state q, in every event-handling round driven by a p-event, the protocol process for α_i performs the steps in Fig. 6b. We do not intend Fig. 6b to convey a real "algorithm"; it should be considered just as a stylized description of what event-handling roughly entails in the distributed approach. Intentionally, also, the description in this figure misses a number of actually essential steps (e.g., for dealing with cases where a protocol process simultaneously receives messages from different protocol processes, which eventually may cause deadlock). Proença developed a distributed algorithm that takes these issues into account [11]; we ignore them here, because they do not matter in the rest of this paper.

Event handler for a big automaton $\alpha = (Q, \mathcal{P}, \longrightarrow, \iota)$ for a connector with ports $\mathcal{P}$ and boundary ports $B \subseteq \mathcal{P}$ in the centralized approach, called every time an event occurs on one of its ports

Input: a port p on which an event occurred, a context $C \subseteq \mathcal{P} \cap B$ of boundary ports with a pending I/O-operation, and a current state q.
Output: t holds the new state of automaton α.
Effect: either an enabled transition fires (if B contained sufficiently many I/O-operations to satisfy that transition's synchronization constraint) and t is set to hold the target state, or all transitions are disabled (otherwise), in which case t holds the current state, q.

```
1. Wake up and let $t$ be $q$.
2. For all transitions $q \xrightarrow{P} q'$, ordered nondeterministically:
   (a) If $p \notin P$, abort this iteration of the loop.
   (b) If $P \cap B \nsubseteq C$ (i.e., not all boundary ports involved in the current transition have a pending I/O-operation), continue (i.e., skip to the next iteration).
   (c) Complete the pending I/O-operations on all ports $P \cap B$ (i.e., fire a transition).
   (d) Let $t$ be $q'$ and abort the loop (i.e., a transition involving $p$ fired, so the $p$-event is successfully handled and no more other transitions need be checked).
3. Go dormant.
```

(a) Centralized approach.

Event handler for a small automaton $\alpha_i = (Q_i, \mathcal{P}_i, \longrightarrow_i, \iota_i)$ for a constituent of a connector with ports $\mathcal{P}$ and boundary ports $B \subseteq \mathcal{P}$ in the distributed approach (such that $\mathcal{P}_i \subseteq \mathcal{P}$), called every time an event occurs on one of its ports

Input: a port p on which an event occurred, a context $C \subseteq \mathcal{P} \cap B$ of boundary ports with a pending I/O-operation, and a current state q.
Output: t holds the new state of automaton α_i.
Effect: either through the firing of enabled local transitions (including a local transition of α_i), an enabled global transition fires (if B contained sufficiently many I/O-operations to satisfy that transition's synchronization constraint) and t is set to hold the target state of the firing local transition in α_i, or all global transitions are disabled (otherwise), in which case t holds the current state of α_i, q.

```
1. Wake up and let $t$ be $q$.
2. Let a set of synchronization constraints $\mathbf{P}_i$ be $\emptyset$.
3. For all transitions $q \xrightarrow{P_i}_i q'$, ordered nondeterministically:
   (a) If $p \notin P_i$, continue (i.e., skip to the next iteration).
   (b) If $P_i \cap B \nsubseteq C$, continue.
   (c) Let a set of synchronization constraints $\mathbf{P}$ be $\emptyset$.
   (d) For all ports $p' \in P_i \setminus B$:
       i. Send a message to the protocol process $j$ that shares access to $p'$ to ask which synchronization constraints must hold for that process to fire a transition involving $p'$.
       ii. Await a message from $j$ with an answer $\mathbf{P}_j$.
       iii. Let $\mathbf{P}$ be $\{P_i \cup P_j \mid P_j \in \mathbf{P}_j\}$.
   (e) Let $\mathbf{P}_i$ be $\mathbf{P}_i \cup \mathbf{P}$.
4. If the event involving $p$ originated from a computation process:
   (a) For all synchronization constraints $P \in \mathbf{P}_i$, ordered nondeterministically:
       i. Complete the pending I/O-operations on all ports $P \cap \mathcal{P}_i \cap B$ (i.e., fire a local transition).
       ii. Send $P$ to all processes sent messages to in Step 3.
       iii. Let $t$ be a state such that $q \xrightarrow{P \cap \mathcal{P}_i}_i t$ and abort the loop.
   (b) If the loop was not manually aborted in Step 4-a-iii, send $\emptyset$ to all processes sent messages to in Step 3.
   Else, if the event involving $p$ originated from a protocol process $h$:
   (a) Send a message to $h$ with an answer $\mathbf{P}_i$.
   (b) Await a message with a synchronization constraint $P$.
   (c) If $P \neq \emptyset$, complete the pending I/O-operations on all ports $P \cap \mathcal{P}_i \cap B$ (i.e., fire a local transition).
   (d) Send $P$ to all processes asked questions to in Step 3.
   (e) If $P \neq \emptyset$, let $t$ be a state such that $q \xrightarrow{P \cap \mathcal{P}_i}_i t$.
5. Go dormant.
```

(b) Distributed approach.

Fig. 6. Event-handling routines for a p-event.

3. Local product operator

Essentially, $\boxtimes$ *propagates synchrony* over successive applications. We explain this through an example. Suppose Alice knows about ports {A, B} and has one transition in which she fires exactly those ports (e.g., automaton α in Fig. 2). Similarly, suppose Bob knows about ports {B, C} and has one transition in which he fires exactly those ports (e.g., automaton β in Fig. 2). Because these two transitions satisfy $\diamondsuit$, the product of Alice and Bob has one transition labeled by {A, B, C} (e.g., the first automaton in Fig. 3). This means that Alice and Bob always synchronize on their shared port B: Alice can perform her transition (i.e., she is *willing* to fire B) only if Bob can perform his (i.e., only if he is *ready* to fire B) and vice versa. Now, suppose Carol knows about ports {C, D} and has one transition in which she fires exactly those ports (e.g., automaton γ

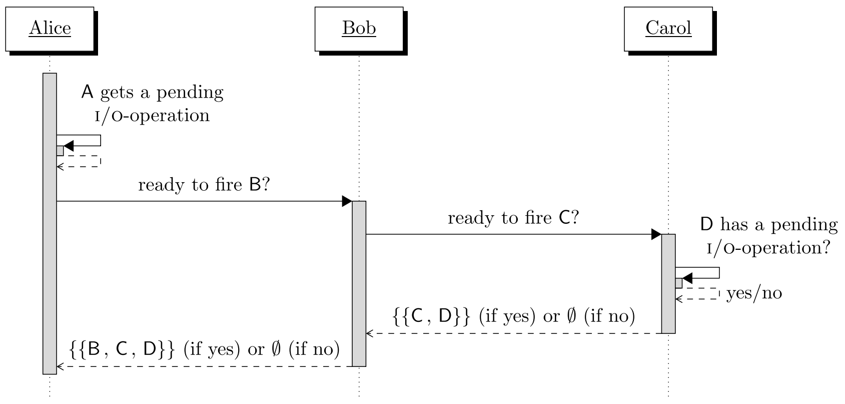
Alice
Bob
Carol
A gets a pending I/O-operation
ready to fire B?
ready to fire C?
D has a pending I/O-operation?
yes/no
{{C, D}} (if yes) or ∅ (if no)
{{B, C, D}} (if yes) or ∅ (if no)

Fig. 7. Sequence diagram for messages exchanged between Alice (i.e., automaton α in Fig. 2), Bob (i.e., automaton β in Fig. 2), and Carol (i.e., automaton γ in Fig. 2) in the distributed approach (cf. Fig. 6b).

in Fig. 2). By the same reasoning as before, the product of [the product of Alice and Bob][5] and Carol has one transition labeled by {A, B, C, D}. Thus, in the product of Alice, Bob, and Carol, Alice "transitively" synchronizes with Carol, through Bob. This property of ⊠ models an important feature of Reo: compositional construction of globally synchronous protocol steps out of locally synchronous parts.

The problem addressed in this paper is that code generators using the centralized approach produce connector implementations that may unnecessarily restrict parallelism. To illustrate this problem, suppose Erin knows about ports {E, F} and has one transition in which she fires exactly those ports (e.g., automaton η in Fig. 2). The product of Alice, Bob, Carol, and Erin computed by a tool using the centralized approach has three transitions (e.g., the fifth automaton in Fig. 3): one labeled by {A, B, C, D} (Alice, Bob, Carol make a transition), another labeled by {E, F} (Erin makes a transition), and yet another labeled by {A, B, C, D, E, F} (Alice, Bob, Carol and Erin coincidentally make a transition at the same time by true concurrency). At run-time, the single protocol process for this big PA nondeterministically picks one of those transitions, checks it for enabledness (i.e., checks whether all ports in its synchronization constraint have a pending I/O-operation), and if so, fires it. Consequently, as soon as the protocol process has selected the transition labeled by {A, B, C, D}, the transition labeled by {E, F} has to wait for the next round, even if it becomes enabled during the current round. In fact, even if the transition labeled by {E, F} was enabled already at the time the protocol process selected the transition labeled by {A, B, C, D}, the former transition has to wait for the next round. In this case, the protocol process might as well have selected the combined "true concurrency transition", labeled by {A, B, C, D, E, F}, but because we do not stipulate maximal progress, the protocol process did not have to; see also the last paragraph in Section 2.1. To summarize, Erin cannot run at her own pace despite being *independent* of Alice, Bob, and Carol.

Although the centralized approach may unnecessarily restrict parallelism, it guarantees high *throughput* compared to the alternative, distributed approach of generating code for Alice, Bob, Carol, and Erin individually. The problem with the distributed approach is the communication necessary for applying ⊠ at run-time. To see this, suppose that we indeed have separate protocol processes for Alice, Bob, Carol, and Erin. Now, if Alice at some point becomes willing to fire her {A, B} transition (i.e., once an I/O-operation has become pending on A), she must ask Bob if he is ready to fire his {B, C} transition. Before he can answer Alice's question, however, Bob in turn must ask Carol if she is ready to fire her {C, D} transition. Fig. 7 shows the messages involved in a sequence diagram. The need for all this communication negatively affects throughput: in practice, because of the inherent overhead involved in sending/receiving/processing messages, it takes much longer for Alice, Bob, and Carol to agree on synchronously executing their individual transitions than for one big PA to make and carry out such a decision by itself.[6] Nevertheless, the distributed approach has high parallelism: Erin can fire her transition *while* Alice, Bob, and Carol communicate to come to an agreement. The centralized and distributed approaches thus force one to choose between two desirable properties: high throughput between *inter*dependent PA, at the cost of parallelism, and high parallelism between *in*dependent ones, at the cost of throughput. As stated in the introduction, we need to find a middle ground between the purely centralized and fully distributed approaches that has both these desirable qualities.

Working toward such an approach, we start from the purely distributed approach of applying ⊠ at run-time through global communication between protocol processes (e.g., Alice talks to Bob, who in turn talks to Carol, etc.). To improve the throughput of this approach, our idea is to bound the globalness of communication: generally, when some Alice asks

[5] Square brackets for readability.

[6] An interesting piece of future work, inspired by one of the reviewers of this paper, is to explicitly represent the overhead of message exchanges involved in a consensus algorithm in the automata semantics. This would enable us to formally quantify and reason in theory, about this overhead and its effect on throughput in practice. One way to do this is to annotate port automata with stochastic information for representing the delays of data-flows, as already done for constraint automata by Baier and Wolf [24]. The finer granularity offered by Action Constraint Automata of Kokash et al. [25] may also be useful to articulate this overhead.

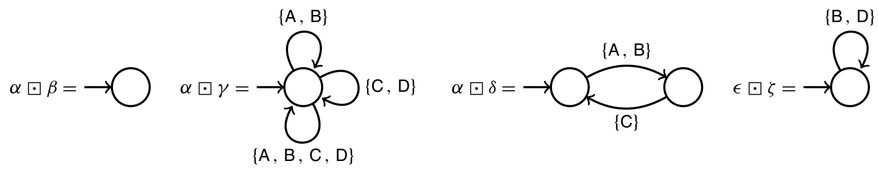
$\alpha \boxdot \beta =$
$\alpha \boxdot \gamma =$
{A , B}
{C , D}
{A , B , C , D}
$\alpha \boxdot \delta =$
{A , B}
{C}
$\epsilon \boxdot \zeta =$
{B , D}

Fig. 8. Automata constructed using $\boxdot$ (α, β, γ, δ, ϵ, and ζ denote the automata in Fig. 2).

some Bob if he is ready to fire a transition involving shared ports, Bob *should* immediately answer *without engaging others*. By doing so, Alice and Bob indeed achieve a higher throughput—less communication means less overhead—while, for the sake of high parallelism, independent others can still run at their own pace. Importantly, however, protocol processes no longer compute the application of $\boxtimes$ in this new approach, simply because the new distributed algorithm they now use does not model $\boxtimes$ anymore. Instead, those processes compute the application of another, new, so far unknown product operator whose implementation requires only local communication. Problematically, however, applying that new product operator instead of $\boxtimes$ can be unsound or incomplete, sometimes to the extent of deadlock. Which of those two happens depends on *how* Bob immediately answers Alice in cases where he actually should have consulted Carol (and possibly others). If Bob replies being ready, the dataflow on Alice's ports (including her ports shared with Bob) incorrectly introduces asynchrony between Bob's two ports. However, if Bob always replies *not* being ready, he and Alice never interact on their shared ports. Clearly, if we want to use the simplified distributed algorithm, we should better understand the potentially severe consequences of doing so. In the rest of this section, we therefore formalize the new product operator that models the simplified distributed algorithm. Then, in the next sections, we study under which circumstances substituting $\boxtimes$ (i.e., the old algorithm) with our new product operator (i.e., the new algorithm) is semantics-preserving.

First, we introduce a relation that defines when transitions of Alice and Bob represent execution steps in which they *strongly agree* on their behavior (cf. Definition 2 of $\Diamond$). In that case, they agree on which of their shared ports to fire (possibly none), and either Alice forbids Bob to simultaneously fire any other port or vice versa. Afterward, we define our new product operator on PA.

Definition 4 *(Strong agreement relation).* The strong agreement relation, denoted by $\blacklozenge$, is the relation on $\wp(\mathbb{P}\text{ORT})^2 \times \wp(\mathbb{P}\text{ORT})^2$ defined as:

$$(\mathcal{P}_\alpha\,,\,P_\alpha)\blacklozenge(\mathcal{P}_\beta\,,\,P_\beta)\ \textbf{iff}\ \left[P_\alpha\subseteq\mathcal{P}_\alpha\ \textbf{and}\ P_\beta\subseteq\mathcal{P}_\beta\ \textbf{and}\ \begin{bmatrix}P_\alpha=\mathcal{P}_\alpha\cap P_\beta\ \textbf{or}\ P_\beta=\mathcal{P}_\beta\cap P_\alpha\\ \textbf{or}\ \mathcal{P}_\alpha\cap P_\beta=\emptyset=\mathcal{P}_\beta\cap P_\alpha\end{bmatrix}\right]$$

Definition 5 *(Local product operator, l-product).* The local product operator, l-product, denoted by $\_\boxdot\_$, is the operator on $\mathbb{P}\text{A}\times\mathbb{P}\text{A}$ defined by the following equation:

$$\alpha\boxdot\beta=(\mathsf{State}(\alpha)\times\mathsf{State}(\beta)\,,\,\mathsf{Port}(\alpha)\cup\mathsf{Port}(\beta)\,,\,\longrightarrow\,,\,(\mathsf{init}(\alpha)\,,\,\mathsf{init}(\beta)))$$

where $\longrightarrow$ denotes the smallest relation induced by INDEPA, INDEPB (see Definition 3), and the following rule:

$$\frac{q_\alpha\xrightarrow{P_\alpha}_\alpha q'_\alpha\ \textbf{and}\ q_\beta\xrightarrow{P_\beta}_\beta q'_\beta\ \textbf{and}\ (\mathsf{Port}(\alpha)\,,\,P_\alpha)\blacklozenge(\mathsf{Port}(\beta)\,,\,P_\beta)}{(q_\alpha\,,\,q_\beta)\xrightarrow{P_\alpha\cup P_\beta}(q'_\alpha\,,\,q'_\beta)}\ (\text{STAGR})$$

Fig. 8 shows examples of the application of $\boxdot$. The following proposition follows immediately from Definition 5.

Proposition 1. $\mathsf{Port}(\alpha\boxdot\beta)=\mathsf{Port}(\alpha)\cup\mathsf{Port}(\beta)$.

The following lemma states that bisimilarity is a congruence.

Lemma 2. *(See Lemma 2, [26].)* $\left[\alpha\approx\beta\ \textbf{and}\ \gamma\approx\delta\right]$ **implies** $\alpha\boxdot\gamma\approx\beta\boxdot\delta$.

Furthermore, $\boxdot$ is commutative up-to bisimilarity, but generally *non*associative. For instance, $(\alpha\boxdot\gamma)\boxdot\beta\not\approx\alpha\boxdot(\gamma\boxdot\beta)$, where α, β, and γ denote the PAin Fig. 2: both l-products have one state, but whereas the former l-product has no transitions, the latter l-product has one transition labeled by {A , B , C , D}. Its nonassociativity makes using $\boxdot$ in practice quite nontrivial. We address this issue in Section 5. To minimize numbers of parentheses in our notation, we assume right-associative notation for $\boxdot$. For instance, we write $\alpha\boxdot\beta\boxdot\gamma\boxdot\delta$ for $\alpha\boxdot(\beta\boxdot(\gamma\boxdot\delta))$.

We define a $\boxdot$-*term* as a word t in the language generated by the grammar in Fig. 9a; we define the *meaning* of $\boxdot$-terms with the interpretation function $[\![\cdot]\!]$ in Fig. 9b.

$$\begin{aligned} \alpha &:= \text{any element from } \mathbb{P}\textsc{a} \\ t &:= \alpha \mid t\ [\,\cdot\,]\ t \end{aligned}$$

(a) Syntax

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![t\ [\,\cdot\,]\ t]\!] &= [\![t]\!] \boxdot [\![t]\!] \end{aligned}$$

(b) Semantics

Fig. 9. $\boxdot$-terms.

4. Substituting $\boxtimes$ with $\boxdot$

4.1. Complete characterization

As informally explained earlier, substituting $\boxtimes$ with $\boxdot$ is not always semantics-preserving. It is, for instance, for the two l-products in the middle of Fig. 8 (cf. the second and the third products in Fig. 3) but not for the l-products on the sides (cf. the first and the fourth products in Fig. 3). In this section and the next, we study under which circumstances substituting $\boxtimes$ with $\boxdot$ is semantics-preserving.

To determine when substituting $\boxtimes$ with $\boxdot$ is semantics-preserving, we define when Alice is a *subautomaton* of Bob. In that case, Bob has at least every transition that Alice has.

Definition 6 *(Subautomaton relation).* The subautomaton relation, denoted by $\sqsubseteq$, is the relation on $\mathbb{P}\textsc{a} \times \mathbb{P}\textsc{a}$ defined as:

$$(Q\,,\,\mathcal{P}\,,\,\longrightarrow_\alpha\,,\,\imath) \sqsubseteq (Q\,,\,\mathcal{P}\,,\,\longrightarrow_\beta\,,\,\imath) \ \textbf{ iff } \ \longrightarrow_\alpha \subseteq \longrightarrow_\beta$$

The following proposition follows directly from the previous definition. In the rest of this section, we investigate under which circumstances its premise holds.

Proposition 2. $\left[\alpha \sqsubseteq \beta \ \textbf{ and } \ \beta \sqsubseteq \alpha\right]$ **implies** $\alpha = \beta$.

Before showing that the l-product of Alice and Bob is a subautomaton of their product, the next lemma states that strong agreement implies weak agreement: if Alice fires exactly those shared ports that Bob fires or vice versa, Alice and Bob agree on their shared ports.

Lemma 3. $(\mathcal{P}_\alpha\,,\,P_\alpha) \blacklozenge (\mathcal{P}_\beta\,,\,P_\beta)$ **implies** $(\mathcal{P}_\alpha\,,\,P_\alpha) \lozenge (\mathcal{P}_\beta\,,\,P_\beta)$.

Proof (Outline). By applying Definition 4 of $\blacklozenge$ and basic set theory to the premise of the lemma, we can derive:

$$\begin{bmatrix} P_\alpha \subseteq \mathcal{P}_\alpha \ \textbf{ and } \ P_\beta \subseteq \mathcal{P}_\beta \\ \textbf{and } \ P_\alpha = \mathcal{P}_\alpha \cap P_\beta \end{bmatrix} \ \textbf{or} \ \begin{bmatrix} P_\alpha \subseteq \mathcal{P}_\alpha \ \textbf{ and } \ P_\beta \subseteq \mathcal{P}_\beta \\ \textbf{and } \ P_\beta = \mathcal{P}_\beta \cap P_\alpha \end{bmatrix} \ \textbf{or} \ \begin{bmatrix} P_\alpha \subseteq \mathcal{P}_\alpha \ \textbf{ and } \ P_\beta \subseteq \mathcal{P}_\beta \\ \textbf{and } \ \mathcal{P}_\alpha \cap P_\beta = \emptyset = \mathcal{P}_\beta \cap P_\alpha \end{bmatrix}$$

By applying basic set theory, we can reduce each of these cases to:

$$P_\alpha \subseteq \mathcal{P}_\alpha \ \textbf{ and } \ P_\beta \subseteq \mathcal{P}_\beta \ \textbf{ and } \ \mathcal{P}_\alpha \cap P_\beta = \mathcal{P}_\beta \cap P_\alpha$$

The lemma subsequently follows from Definition 2 of $\lozenge$. See [27, Lemma 3] for a detailed proof. □

The next lemma states that the l-product of Alice and Bob is a subautomaton of their product: the product of Alice and Bob can do *at least* the same as their l-product.

Lemma 4. $\alpha \boxdot \beta \sqsubseteq \alpha \boxtimes \beta$.

Proof (Outline). Let $\longrightarrow_{\boxdot}$ and $\longrightarrow_{\boxtimes}$ denote the transition relations of $\alpha \boxdot \beta$ and $\alpha \boxtimes \beta$. Suppose $(q_\alpha\,,\,q_\beta) \xrightarrow{P}_{\boxdot} (q'_\alpha\,,\,q'_\beta)$. By applying Definition 5 of $\boxdot$, we can derive that this transition was induced by rule WkAgr, by rule IndepA, or by rule IndepB. In the latter two cases, exactly the same transition exists also in $\alpha \boxtimes \beta$, because Definition 3 of $\boxtimes$ states that IndepA and IndepB induce also $\longrightarrow_{\boxtimes}$. In the former case, by applying the definition of WkAgr, we can derive that a transition $q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha$ in α and a transition $q_\beta \xrightarrow{P_\beta}_\beta q'_\beta$ in β exist such that $P = P_\alpha \cup P_\beta$ and $(\mathsf{Port}(\alpha)\,,\,P_\alpha) \blacklozenge (\mathsf{Port}(\beta)\,,\,P_\beta)$. Then, by applying Lemma 3, we can derive $(\mathsf{Port}(\alpha)\,,\,P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\,P_\beta)$. Then, by applying Definition 3 of $\boxtimes$, we can derive $(q_\alpha\,,\,q_\beta) \xrightarrow{P}_{\boxtimes} (q'_\alpha\,,\,q'_\beta)$. The lemma subsequently follows from basic set theory and Definition 6 of $\sqsubseteq$. See [27, Lemma 4] for a detailed proof. □

The product of Alice and Bob is not necessarily a subautomaton of their l-product: if Alice and Bob agree on which of their shared ports to fire, this does not necessarily mean that they fire no other ports. To characterize the cases in which they do, we define *conditional strong agreement* as a relation "in between" of $\blacklozenge$ and $\lozenge$ (and lifted from transitions to pa):

Alice and Bob conditionally strongly agree iff, for each of their transitions, their weak agreement on which of their shared ports to fire implies their strong agreement.

Definition 7 *(Conditional strong agreement relation).* The conditional strong agreement relation, denoted by $\langle\!\blacklozenge\!\rangle$, is the relation on $\mathbb{P}\mathrm{A} \times \mathbb{P}\mathrm{A}$ defined as:

$$(Q_\alpha\,,\ \mathcal{P}_\alpha\,,\ \longrightarrow_\alpha\,,\ \iota_\alpha)\ \langle\!\blacklozenge\!\rangle\ (Q_\beta\,,\ \mathcal{P}_\beta\,,\ \longrightarrow_\beta\,,\ \iota_\beta)\ \textbf{ iff }\ \left[\begin{array}{c}\left[\begin{array}{l}\left[\begin{array}{c} q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha \textbf{ and } q_\beta \xrightarrow{P_\beta}_\beta q'_\beta \textbf{ and} \\ (\mathrm{Port}(\alpha)\,,\ P_\alpha) \lozenge (\mathrm{Port}(\beta)\,,\ P_\beta) \end{array}\right] \\ \textbf{implies } (\mathrm{Port}(\alpha)\,,\ P_\alpha) \blacklozenge (\mathrm{Port}(\beta)\,,\ P_\beta) \end{array}\right] \\ \textbf{for all } q_\alpha\,,\ q_\beta\,,\ q'_\alpha\,,\ q'_\beta\,,\ P_\alpha\,,\ P_\beta \end{array}\right]$$

The next lemma states that if Alice and Bob conditionally strongly agree, their product is a subautomaton of their l-product (cf. Lemma 4).

Lemma 5. $\alpha \langle\!\blacklozenge\!\rangle \beta$ **implies** $\alpha \boxtimes \beta \sqsubseteq \alpha \boxdot \beta$.

Proof (Outline). Structurally, the proof is similar to the proof of Lemma 4. The main difference is the case where a transition $q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha$ in α and a transition $q_\beta \xrightarrow{P_\beta}_\beta q'_\beta$ in β exist such that $P = P_\alpha \cup P_\beta$ and $(\mathrm{Port}(\alpha)\,,\ P_\alpha) \lozenge (\mathrm{Port}(\beta)\,,\ P_\beta)$. In that case, by applying Definition 7 of $\langle\!\blacklozenge\!\rangle$ to the premise of the lemma, we can derive that for this α and this β, weak agreement ($\lozenge$) implies strong agreement ($\blacklozenge$). The rest of the proof is similar. See [27, Lemma 5] for a detailed proof. □

We end this section with the following theorem: if Alice and Bob conditionally strongly agree, substituting $\boxtimes$ with $\boxdot$ is semantics-preserving (in fact, not just under bisimilarity but even under structural equality).

Theorem 1. $\alpha \langle\!\blacklozenge\!\rangle \beta$ **implies** $\alpha \boxdot \beta = \alpha \boxtimes \beta$.

Proof (Outline). The theorem follows from Lemmas 4 and 5 and Proposition 2. See [27, Theorem 1] for a detailed proof. □

4.2. *Cheaper characterization*

To test if Alice and Bob conditionally strongly agree, one must pairwise compare their transitions. This can be computationally expensive (i.e., $\mathcal{O}(n_1 n_2)$, where n_1 and n_2 denote the numbers of transitions), and it makes the $\langle\!\blacklozenge\!\rangle$-based characterization, although (conjectured to be) complete, hard to apply in practice. Computational complexity aside, the $\langle\!\blacklozenge\!\rangle$-based characterization does not give us any insight into how to obtain, from the distributed-approach-with-simplified-algorithm, the middle ground that we set out to find. In this section, we study a cheaper characterization of (a subset of) conditionally strongly agreeing PA that does give us such insight: by the end of this section, we can derive our middle ground from the theory developed up to that point.

In Section 3, we explained reduced parallelism as a disadvantage of centralized implementations in terms of *independence*. Therefore, substituting $\boxtimes$ with $\boxdot$ should be semantics-preserving at least when applied to such independent PA. We start by formally defining when Alice and Bob are independent: in that case, they have no shared ports.

Definition 8 *(Independence relation).* The independence relation, denoted by $\asymp$, is the relation on $\mathbb{P}\mathrm{A} \times \mathbb{P}\mathrm{A}$ defined as:

$$\alpha \asymp \beta \ \textbf{ iff }\ \mathrm{Port}(\alpha) \cap \mathrm{Port}(\beta) = \emptyset$$

The following lemma states that if Alice and Bob are independent, they conditionally strongly agree (because their independence means that Alice and Bob have no shared ports).

Lemma 6. $\alpha \asymp \beta$ **implies** $\alpha \langle\!\blacklozenge\!\rangle \beta$.

Proof (Outline). Suppose $(\mathrm{Port}(\alpha)\,,\ P_\alpha) \lozenge (\mathrm{Port}(\beta)\,,\ P_\beta)$ (i.e., part of the premise in Definition 7 of $\langle\!\blacklozenge\!\rangle$). Then, by applying Definition 2 of $\lozenge$ and by applying Definition 8 of $\asymp$ to the premise of the lemma, we can derive $\left[\mathrm{Port}(\alpha) \cap \mathrm{Port}(\beta) = \emptyset \textbf{ and } \left[P_\alpha \subseteq \mathrm{Port}(\alpha) \textbf{ and } P_\beta \subseteq \mathrm{Port}(\beta) \textbf{ and } \mathrm{Port}(\alpha) \cap P_\beta = \mathrm{Port}(\beta) \cap P_\alpha\right]\right]$. Then, by applying basic set theory and Definition 4 of $\blacklozenge$, we can derive $(\mathrm{Port}(\alpha)\,,\ P_\alpha) \blacklozenge (\mathrm{Port}(\beta)\,,\ P_\beta)$. The lemma subsequently follows from Definition 7 of $\langle\!\blacklozenge\!\rangle$. See [27, Lemma 6] for a detailed proof. □

Lemma 6 and Theorem 1 imply that substituting $\boxtimes$ with $\boxdot$ is semantics-preserving if their operands satisfy the independence relation. Moreover, checking $\asymp$ costs less than checking whether PA conditionally strongly agree, namely $\mathcal{O}(1)$ instead of $\mathcal{O}(n_1 n_2)$. The following lemma states another important property, namely that $\boxdot$ preserves independence: if Alice is independent of both Bob and Carol individually, she is independent of Bob and Carol together. The corollary following this lemma generalizes this result from two to k PA.

Lemma 7. $\big[\alpha \asymp \beta \;\textbf{ and }\; \alpha \asymp \gamma\big] \;\textbf{ implies }\; \alpha \asymp \beta \boxdot \gamma$.

Proof (Outline). By applying Definition 8 of $\asymp$ and basic set theory to the premise of the lemma, we can derive $\mathsf{Port}(\alpha) \cap (\mathsf{Port}(\beta) \cup \mathsf{Port}(\gamma)) = \emptyset$. The lemma subsequently follows from Proposition 1 and Definition 8 of $\asymp$. See [27, Lemma 7] for a detailed proof. □

Corollary 1. $\big[\alpha \asymp \beta_1 \;\textbf{ and }\; \cdots \;\textbf{ and }\; \alpha \asymp \beta_k\big] \;\textbf{ implies }\; \alpha \asymp (\beta_1 \boxdot \cdots \boxdot \beta_k)$.

Although checking PA for independence is cheap, the result implied by Lemma 6 and Theorem 1 in its present form has limited practical value: total independence is a condition rarely satisfied by the PA encountered in real-world compilation scenarios. To get a more useful similar result, we now introduce the notion of *slavery* and afterward combine it with independence. We start by formally defining when Bob is a slave of Alice: in that case, every transition of Bob that involves *some* ports shared with Alice, involves *only* ports shared with Alice. In other words, if shared ports are involved, Alice completely dictates what Bob does. Our notion of slavery does not prevent Bob from freely executing transitions involving only ports that Alice does not know about, which may allow Bob to be a slave of PA other than Alice as well.

Definition 9 *(Slave relation).* The slave relation, denoted by $\mapsto$, is the relation on $\mathbb{P}\mathrm{A} \times \mathbb{P}\mathrm{A}$ defined as:

$$(Q_\beta\,,\, \mathcal{P}_\beta\,,\, \longrightarrow_\beta\,,\, \imath_\beta) \mapsto \alpha \;\textbf{ iff }\; \left[\left[\begin{bmatrix} q_\beta \xrightarrow{P_\beta} q'_\beta \;\textbf{ and} \\ P_\beta \cap \mathsf{Port}(\alpha) \neq \emptyset \end{bmatrix} \;\textbf{ implies }\; P_\beta \subseteq \mathsf{Port}(\alpha)\right] \;\textbf{ for all }\; q_\beta\,,\, q'_\beta\,,\, P_\beta\right].$$

The following lemma states that if Bob is a slave of Alice, they conditionally strongly agree (i.e., Alice forces her will upon Bob).

Lemma 8. $\beta \mapsto \alpha \;\textbf{ implies }\; \beta \,\lozenge\!\!\!\blacklozenge\, \alpha$.

Proof (Outline). Suppose $\big[q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha \;\textbf{ and }\; q_\beta \xrightarrow{P_\beta}_\beta q'_\beta \;\textbf{ and }\; (\mathsf{Port}(\alpha)\,,\, P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\, P_\beta)\big]$ (i.e., the premise in Definition 7 of $\lozenge\!\!\!\blacklozenge$). Then, by applying Definition 9 of $\mapsto$ and basic set theory to the premise of the lemma, we can derive:

$$\begin{aligned} &\big[(\textbf{not }\; q_\alpha \xrightarrow{P_\alpha} q'_\alpha) \;\textbf{ and }\; q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha \;\textbf{ and }\; q_\beta \xrightarrow{P_\beta}_\beta q'_\beta \;\textbf{ and }\; (\mathsf{Port}(\alpha)\,,\, P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\, P_\beta)\big] \\ \textbf{or }\; &\big[P_\alpha \cap \mathsf{Port}(\beta) = \emptyset \;\textbf{ and }\; q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha \;\textbf{ and }\; q_\beta \xrightarrow{P_\beta}_\beta q'_\beta \;\textbf{ and }\; (\mathsf{Port}(\alpha)\,,\, P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\, P_\beta)\big] \\ \textbf{or }\; &\big[P_\alpha \subseteq \mathsf{Port}(\beta) \;\textbf{ and }\; q_\alpha \xrightarrow{P_\alpha}_\alpha q'_\alpha \;\textbf{ and }\; q_\beta \xrightarrow{P_\beta}_\beta q'_\beta \;\textbf{ and }\; (\mathsf{Port}(\alpha)\,,\, P_\alpha) \lozenge (\mathsf{Port}(\beta)\,,\, P_\beta)\big] \end{aligned}$$

The first case immediately reduces to **false**. For the other two cases, by applying Definition 2 of $\lozenge$ and basic set theory, we can derive:

$$\begin{bmatrix} P_\alpha \subseteq \mathsf{Port}(\alpha) \;\textbf{ and }\; P_\beta \subseteq \mathsf{Port}(\beta) \\ \textbf{and }\; \mathsf{Port}(\alpha) \cap P_\beta = \emptyset = \mathsf{Port}(\beta) \cap P_\alpha \end{bmatrix} \;\textbf{ or }\; \begin{bmatrix} P_\alpha \subseteq \mathsf{Port}(\alpha) \;\textbf{ and }\; P_\beta \subseteq \mathsf{Port}(\beta) \\ \textbf{and }\; \mathsf{Port}(\alpha) \cap P_\beta = \mathsf{Port}(\beta) \cap P_\alpha \end{bmatrix}$$

Both cases now reduce to $(\mathsf{Port}(\alpha)\,,\, P_\alpha) \blacklozenge (\mathsf{Port}(\beta)\,,\, P_\beta)$ by applying Definition 4 of $\blacklozenge$. The lemma subsequently follows from Definition 7 of $\lozenge\!\!\!\blacklozenge$. See [27, Lemma 8] for a detailed proof. □

Lemma 8 and Theorem 1 imply that substituting $\boxtimes$ with $\boxdot$ is semantics-preserving if their operands satisfy the slave relation. Moreover, checking $\mapsto$ costs less than checking whether PA conditionally strongly agree, namely $\mathcal{O}(n_1)$ instead of $\mathcal{O}(n_1 n_2)$. The following lemma states another important property, namely that $\boxdot$ preserves slavery: if Bob is a slave of Alice, and if he additionally is independent or a slave of Carol, he is a slave of Alice and Carol together.

Lemma 9. $\big[\beta \mapsto \alpha \;\textbf{ and }\; \big[\beta \asymp \gamma \;\textbf{ or }\; \beta \mapsto \gamma\big]\big] \;\textbf{ implies }\; \beta \mapsto \alpha \boxdot \gamma$.

Proof (Outline). Suppose $\big[q_\beta \xrightarrow{P_\beta} q'_\beta \;\textbf{ and }\; P_\beta \cap \mathsf{Port}(\alpha \boxdot \gamma) \neq \emptyset\big]$ (i.e., the premise in Definition 9 of $\mapsto$). Then, by applying Proposition 1 and basic set theory, we can derive:

$$\big[q_\beta \xrightarrow{P_\beta} q'_\beta \;\textbf{ and }\; P_\beta \cap \mathsf{Port}(\alpha) \neq \emptyset\big] \;\textbf{ or }\; \big[q_\beta \xrightarrow{P_\beta} q'_\beta \;\textbf{ and }\; P_\beta \cap \mathsf{Port}(\gamma) \neq \emptyset\big]$$

In the former case, by applying Definition 9 of $\mapsto$ to $\beta \mapsto \alpha$, we can derive $P_\beta \subseteq \text{Port}(\alpha)$. Then, by applying basic set theory and Proposition 1, we can derive $P_\beta \subseteq \text{Port}(\alpha \boxdot \beta)$. In the latter case, we distinguish two cases based on the premise of the lemma. If $\beta \asymp \gamma$, the latter case reduces to **false** by applying Definition 8 of $\asymp$ to $\beta \asymp \gamma$, by observing $P_\beta \subseteq \text{Port}(\beta)$, and by applying basic set theory. Otherwise, if $\beta \mapsto \gamma$, the latter case reduces to $P_\beta \subseteq \text{Port}(\alpha \boxdot \beta)$ as in the former case. Either way, we can derive $P_\beta \subseteq \text{Port}(\alpha \boxdot \beta)$. The lemma subsequently follows from Definition 9 of $\mapsto$.

(The premise of this lemma differs from the premise of [27, Lemma 9], which does not contain the conjunct $\left[\beta \asymp \gamma \textbf{ or } \beta \mapsto \gamma\right]$. The reason is that [27, Lemma 9] in fact does not hold, and its proof contains a mistake. By adding $\left[\beta \asymp \gamma \textbf{ or } \beta \mapsto \gamma\right]$ to the premise, we could fix the proof, and fortunately, the only place where this lemma is used satisfies the updated premise, namely the proof of Lemma 11, below.) □

By combining independence and slavery, we obtain the notion of *conditional slavery*: Bob is a conditional slave of Alice iff Alice and Bob not being independent implies that Bob is a slave of Alice.

Definition 10 *(Conditional slave relation).* The conditional slave relation, denoted by $\Mapsto$, is the relation on $\mathbb{P}\text{A} \times \mathbb{P}\text{A}$ defined as:

$$\beta \Mapsto \alpha \textbf{ iff } \left[\beta \asymp \alpha \textbf{ or } \beta \mapsto \alpha\right]$$

The following lemma states that if Bob is a conditional slave of Alice, they conditionally strongly agree (i.e., Alice and Bob are independent or Alice forces her will upon Bob).

Lemma 10. $\beta \Mapsto \alpha$ **implies** $\beta \blacklozenge \alpha$.

Proof (Outline). The lemma follows from Definition 10 and Lemmas 6 and 8. See [27, Lemma 10] for a detailed proof. □

The combination of Lemma 10 and Theorem 1 implies that substituting $\boxtimes$ with $\boxdot$ is semantics-preserving if the PA involved satisfy the conditional slave relation. Moreover, checking the conditional slave relation costs the same as checking the slave relation (i.e., less than checking whether PA conditionally strongly agree). The following lemma states another important property, namely that $\boxdot$ preserves conditional slavery: if Bob is a conditional slave of both Alice and Carol individually, he is a conditional slave of Alice and Carol together. The corollary following this lemma generalizes this result from two to k PA.

Lemma 11. $\left[\beta \Mapsto \alpha \textbf{ and } \beta \Mapsto \gamma\right]$ **implies** $\beta \Mapsto \alpha \boxdot \gamma$.

Proof (Outline). The lemma follows from Definition 10 and Lemmas 7 and 9. See [27, Lemma 10] for a detailed proof. □

Corollary 2. $\left[\beta \Mapsto \alpha_1 \textbf{ and } \cdots \textbf{ and } \beta \Mapsto \alpha_k\right]$ **implies** $\beta \Mapsto (\alpha_1 \boxdot \cdots \boxdot \alpha_k)$.

With conditional slavery, in contrast to independence alone, one can characterize a sufficiently large class of PA that satisfies the premise of Theorem 1 (i.e., for which substituting $\boxtimes$ with $\boxdot$ is semantics-preserving), as follows. Suppose that we have a list of k PA such that every i-th PA in the list is a conditional slave of all PA in a higher position (the order matters here, because $\boxdot$ is nonassociative). Then, the l-product of all PA in this list, starting from the ones with the highest positions and working our way down, is in the class. The following definition formalizes this. Recall that we adopted right-associative notation for $\boxdot$.

Definition 11. $\mathcal{A} \subseteq \mathbb{P}\text{A}$ denotes the smallest set induced by the following rule:

$$\frac{\left[i \neq j \textbf{ implies } \alpha_i \Mapsto \alpha_j\right] \textbf{ for all } 1 \leq i < j \leq k}{\alpha_1 \boxdot \cdots \boxdot \alpha_k \in \mathcal{A}}$$

Henceforth, instead of writing $\alpha_1 \boxdot \cdots \boxdot \alpha_k$, we sometimes write $\alpha_1 \cdots \alpha_k$ or, even more compactly, $[\alpha]_1^k$.[7]

The following theorem states that for every PA in $\mathcal{A}$, substituting $\boxtimes$ for $\boxdot$ is semantics-preserving.

Theorem 2. $\alpha_1 \boxdot \cdots \boxdot \alpha_k \in \mathcal{A}$ **implies** $\alpha_1 \boxdot \cdots \boxdot \alpha_k = \alpha_1 \boxtimes \cdots \boxtimes \alpha_k$.

[7] Mixing these notations does not lead to insertion of parentheses: the right-associative notation that we adopted for $\boxdot$ is preserved. For instance, $[\alpha]_1^3\beta$ stands for $\alpha_1 \boxdot (\alpha_2 \boxdot (\alpha_3 \boxdot \beta))$—not for $(\alpha_1 \boxdot (\alpha_2 \boxdot \alpha_3)) \boxdot \beta$.

Proof. We prove the lemma by induction on k. The base case, $k = 1$ follows straightforwardly. In the inductive step, $k > 1$, we assume that the lemma holds for all $k' < k$ as our induction hypothesis. By applying Definition 11 of $\mathcal{A}$ to the premise of the lemma, we can derive:

$$\big[i \neq j \textbf{ implies } \alpha_i \Mapsto \alpha_j\big] \textbf{ for all } 1 \leq i < j \leq k$$

Then, by pulling out $i = 1$, we can derive:

$$\alpha_1 \Mapsto \alpha_2\,,\,\ldots\,,\,\alpha_k \textbf{ and } \big[\big[i \neq j \textbf{ implies } \alpha_i \Mapsto \alpha_j\big] \textbf{ for all } 2 \leq i < j \leq k\big]$$

Then, by applying Corollary 2 to the left conjunct and the induction hypothesis to the right conjunct, we can derive $\big[\alpha_1 \Mapsto (\alpha_2 \boxdot \cdots \boxdot \alpha_k) \textbf{ and } \alpha_2 \boxdot \cdots \boxdot \alpha_k = \alpha_2 \boxtimes \cdots \boxtimes \alpha_k\big]$. Then, by applying Lemma 10 and Theorem 1 to the left conjunct, we can derive:

$$\alpha_1 \boxdot \cdots \boxdot \alpha_k = \alpha_1 \boxtimes (\alpha_2 \boxdot \cdots \boxdot \alpha_k) \textbf{ and } \alpha_2 \boxdot \cdots \boxdot \alpha_k = \alpha_2 \boxtimes \cdots \boxtimes \alpha_k$$

The lemma subsequently follows from substituting the right conjunct into the left conjunct. See [27, Theorem 2] for a detailed proof. □

Although $\alpha_1 \boxdot \cdots \boxdot \alpha_k = \alpha_1 \boxtimes \cdots \boxtimes \alpha_k$ generally does not imply $\alpha_1 \boxdot \cdots \boxdot \alpha_k \in \mathcal{A}$, it does for the examples considered in this paper. For instance, Figs. 3 and 8 show that $\beta \boxdot \delta = \beta \boxtimes \delta$ (Fig. 2 defines β and δ). By the commutativity of $\boxdot$ and $\boxtimes$, we have also $\delta \boxdot \beta \approx \delta \boxtimes \beta$. Now, because δ is a slave of β, we conclude that $\delta \boxdot \beta$ is an element of $\mathcal{A}$: indeed, if δ makes a transition involving ports shared with β (only B), it fires no other ports (β, in contrast, does fire another port in that case, namely C).

Previously, in the beginning of this section, we claimed that we can derive the middle ground between the distributed and the centralized approach that we set out to find from the theory that we developed so far. We end this section by substantiating that claim. We start by introducing a further restricted class of PA with a more natural interpretation in our context.

Definition 12. $\mathcal{B}$ denotes the smallest set induced by the following rule:

$$\frac{\begin{array}{c}\big[\big[i_1 \neq i_2 \textbf{ implies } \alpha_{i_1} \Mapsto \alpha_{i_2}\big] \textbf{ for all } 1 \leq i_1\,,\,i_2 \leq k\big] \textbf{ and}\\ \big[\big[j_1 \neq j_2 \textbf{ implies } \beta_{j_1} \asymp \beta_{j_2}\big] \textbf{ for all } 1 \leq j_1\,,\,j_2 \leq l\big] \textbf{ and}\\ \big[\alpha_i \Mapsto \beta_j \textbf{ for all } 1 \leq i \leq k\,,\,1 \leq j \leq l\big]\end{array}}{\alpha_1 \boxdot \cdots \boxdot \alpha_k \boxdot \beta_1 \boxdot \cdots \boxdot \beta_l \in \mathcal{B}}$$

The following proposition follows directly from the previous definition.

Proposition 3. $\mathcal{B} \subseteq \mathcal{A}$.

The combination of Proposition 3 and Theorem 2 implies that substituting $\boxtimes$ with $\boxdot$ is semantics-preserving for every PA in $\mathcal{B}$.

Informally, every PA in $\mathcal{B}$ is the l-product of (i) k PA that are conditional slaves of *all* other PA in the term and (ii) l pairwise independent PA that are "masters" of the k conditional slaves. The masters, being pairwise independent, do not *directly* communicate with each other. However, when two or more masters share the same slave (the definition of $\mathcal{B}$ allows this), communication between those masters occurs *indirectly* through that slave. Such indirect communication is always asynchronous: if it were synchronous, the slave involved would fire ports of more than one of its masters in the same transition, which slavery forbids. This interpretation of masters and slaves corresponds exactly to the notion of synchronous and asynchronous *regions* in the Reo literature, perhaps first mentioned by Clarke et al. [28]. Roughly, one can always split a connector into subconnectors—the regions—such that interaction on ports in such a subconnector is either purely asynchronous (i.e., at most one port participates in every firing transition) or requires some synchronization (i.e., more than one port participates in at least one firing transition).

The synchronous regions of a connector are maximal in the sense that no two synchronous regions have shared ports: all synchronous regions are, by definition, pairwise independent. Two subconnectors are "pairwise independent" iff the subgraphs for those subconnectors are disconnected. "Maximality" means that two synchronous regions can never be graph-theoretically adjacent: if they were, neither of these synchronous regions is maximal, because there exists a larger synchronous region, namely the one containing both of them. Because synchronous regions are pairwise independent, the PA for the l synchronous regions of a connector can act as the l masters in the definition of $\mathcal{B}$. The asynchronous regions form the "glue" between the synchronous regions: the PA for every asynchronous region has the same shape as δ in Fig. 2,[8] and consequently, they can act as the k conditional slaves in the definition of $\mathcal{B}$.

[8] Port automaton δ in Fig. 2 describes the behavior of an asynchronous Reo primitive, called Fifo [1,2], with a buffer (of capacity 1) that accepts data on one port (i.e., B), buffers it, and at a later time dispenses that same data on another port (i.e. C). Of the currently common Reo primitives, only Fifo is

To more precisely formalize (a)synchronous regions in terms of PA, let A denote a set of small PA, one for every constituent of the connector under study. The asynchronous regions $\mathcal{R}_{\text{async}}$ of this connector are represented by singleton subsets of A, each of which contains a PA whose every transition has a singleton synchronization constraint:

$$\mathcal{R}_{\text{async}} = \{\{\alpha\} \mid \alpha \in A \ \textbf{and} \ \left[\left[(q\,,\, P\,,\, q') \in \mathsf{Trans}(\alpha) \ \textbf{implies} \ |P| = 1\right] \ \textbf{for all} \ q\,,\, q'\,,\, P\right]\}$$

The requirement of singleton synchronization constraints implies that a PA for a synchronous region never synchronizes any of its ports. Let $B = A \setminus \bigcup \mathcal{R}_{\text{async}}$ denote the set of remaining PA, after computing asynchronous regions. The synchronous regions $\mathcal{R}_{\text{sync}}$, then, are represented by maximal subsets of B (in the sense of graph-connectedness as previously explained):

$$\mathcal{R}_{\text{sync}} = \{f_B(\alpha) \mid \alpha \in B\}$$

where $f_B(\alpha)$ is the smallest set induced by the following rules:

$$\frac{}{\alpha \in f_B(\alpha)} \qquad \frac{\alpha' \in f_B(\alpha) \ \textbf{and} \ \alpha'' \in B \ \textbf{and} \ \mathsf{Port}(\alpha') \cap \mathsf{Port}(\alpha'') \neq \emptyset}{\alpha'' \in f_B(\alpha)}$$

Note that $f_B(\alpha)$ is the same for every α in the same synchronous region (i.e., the previous definition of $\mathcal{R}_{\text{sync}}$ relies on the property that mathematical sets contain no duplicates). It is straightforward to prove that any PA β with only singleton synchronization constraints is a slave of any other PA α with which it shares ports (because in that case, $P_\beta \subseteq \mathsf{Port}(\alpha)$ always holds as required by Definition 9). Thus, every asynchronous region in this formalization indeed corresponds to a conditional slave as stated above. Similarly, it is straightforward to prove that all (products of PA for) synchronous regions are pairwise independent (in the sense of Definition 8). Thus, every synchronous region in this formalization indeed corresponds to a master.

The analysis in the previous paragraphs gives rise to the following partially-distributed/partially-centralized *hybrid approach* to implementing connectors (i.e., the middle-ground that we set out to find), already predicted by Proença et al. [11, 13]. The right branch in Fig. 5 shows the compilation steps involved in generating hybrid implementations of Reo connectors. As in the centralized and the distributed approach, on input of a connector Connector, the compiler first finds a $\boxtimes$-term $\alpha_1 \ [\times] \ \ldots \ [\times] \ \alpha_n$ that models Connector's behavior. Contrasting the centralized and distributed approaches, however, the compiler subsequently determines the (a)synchronous regions of Connector, computes the meaning of each of those regions, and composes the resulting $m \leq n$ "medium" PA into an equivalent $\boxdot$-term $\alpha'_1 \ [\,\cdot\,] \ \ldots \ [\,\cdot\,] \ \alpha'_m$. (To be precise, the compiler first uses associativity and commutativity of $\boxtimes$ to rewrite $\alpha_1 \ [\times] \ \ldots \ [\times] \ \alpha_n$ to a $\boxtimes$-term in which the PA of all regions occur together. Subsequently, the compiler computes the meaning of every region's subterm, which strictly speaking yields a $\boxtimes$-term of m medium PA—not a $\boxdot$-term. However, because the PA for the regions satisfy the definition of $\mathcal{B}$, as argued for above, the compiler can equivalently consider a $\boxdot$-term of the same m medium PA.) Finally, the compiler translates this $\boxdot$-term to m pieces of sequential event-handling code, one for every medium PA in the $\boxdot$-term. Consequently, a Reo-based concurrent application implemented according to the hybrid approach has as many protocol processes as Connector has regions. As in the purely distributed approach, protocol processes for medium PA exchange messages as part of a distributed algorithm for synchronizing their transitions. These message exchanges constitute the simplified distributed algorithm (with only local instead of global communication; see Section 3) that implements the l-product operators between the PA in the $\boxdot$-term, thereby computing the application of those l-product operators at run-time. Importantly, because $\boxdot$ forbids global communication by its construction, this distributed algorithm requires significantly less communication than the original distributed algorithm that implements $\boxtimes$.

In summary: a compiler can always process the set of small PA collectively modeling a connector to a form that satisfies $\mathcal{B}$, by applying $\boxtimes$ between interdependent PA belonging to the same synchronous region at compile-time (for the sake of throughput), and by applying $\boxdot$ between the resulting medium PA plus the PA for the asynchronous regions at run-time (for the sake of parallelism) using the simplified distributed algorithm. Proposition 3 and Theorem 2 ensure that this is semantics-preserving.

5. Associativity

As already briefly remarked, while $\boxtimes$ is associative up-to bisimilarity, $\boxdot$ is not. Thus, although we showed that it does not matter whether we compositionally construct PA in $\mathcal{A}$ (and $\mathcal{B}$) by applying $\boxtimes$ or by applying $\boxdot$—both operators yield the same PA (see Theorem 2)—the fact that $\boxdot$ is nonassociative somehow seems suspicious. In this section, we make this suspicion precise. We start by analyzing the role of $\boxtimes$'s associativity in the context of the distributed approach. This provides us a framework of *huge automata* that we subsequently can use to explain and compensate for the consequences of the nonassociativity of $\boxdot$.

asynchronous, and so, only Fifo instances induce asynchronous regions in the current practice. In general, a PA modeling an asynchronous region can have more than two states or ports but, crucially, each of its transitions has a singleton set of ports as label (as does δ), which guarantees that PA can act as a conditional slave in a $\mathcal{B}$-term.

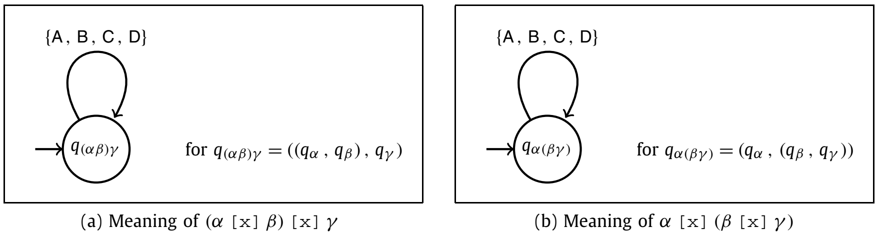
{A , B , C , D}
$q_{(\alpha\beta)\gamma}$
for $q_{(\alpha\beta)\gamma} = ((q_\alpha\,,\, q_\beta)\,,\, q_\gamma)$
(a) Meaning of $(\alpha$ [x] $\beta)$ [x] γ
{A , B , C , D}
$q_{\alpha(\beta\gamma)}$
for $q_{\alpha(\beta\gamma)} = (q_\alpha\,,\, (q_\beta\,,\, q_\gamma))$
(b) Meaning of α [x] $(\beta$ [x] $\gamma)$

Fig. 10. Meanings of α [x] $(\beta$ [x] $\gamma)$ and α [x] $(\beta$ [x] $\gamma)$.

5.1. Distributed approach

Recall that in distributed implementations of Reo connectors, we have a number of protocol processes, each of which executes the event-handling routine in Fig. 6b every time an event occurs (on one the ports it has access to). Perhaps almost obviously, we do not restrict the order in which such protocol processes can execute this routine and, as part of it, exchange messages with each other: every protocol process runs at its own pace, communicating just whenever necessary. Interestingly, however, one can argue that by not placing restrictions on their communication order, those processes *strictly speaking* do not correctly implement (the meaning of) the ⊠-term α_1 [x] $\cdots$ [x] α_n in Fig. 5. To explain this, for the sake of argument, suppose $n = 3$ such that our notation α_1 [x] $\cdots$ [x] α_3 stands either for the ⊠-term $t_1 = (\alpha_1$ [x] $\alpha_2)$ [x] α_3 or for the ⊠-term $t_2 = \alpha_1$ [x] $(\alpha_2$ [x] $\alpha_3)$. Let Alice, Bob, and Carol be names for α_1, α_2, and α_3. Formally, the parentheses in t_1 and t_2 fix the order in which one must apply product operators when computing those terms' meaning, namely from inner to outer subterms. Because the messages exchanged between protocol processes at run-time essentially implement the application of product operators to PA, *strictly speaking*, the order in which message exchanges takes place consequently matters. For instance, in a strict implementation of $(\alpha_1$ [x] $\alpha_2)$ [x] α_3, because Alice and Bob are most deeply nested, only Alice or Bob can start communicating by sending a message to Bob or Alice. Only afterward, $\big[$Alice and Bob$\big]$—with one of them as their representative—and Carol can communicate with each other (cf. Fig. 7). In contrast, in a strict implementation of α_1 [x] $(\alpha_2$ [x] $\alpha_3)$, only Bob or Carol can start communicating by sending a message to Carol or Bob, and only afterward, $\big[$Bob and Carol$\big]$ and Alice can communicate with each other.

A good reason exists for why the literature on distributed Reo implementations has not recorded the previous subtle point: it is immaterial. After all, ⊠ is associative. *Intuitively*, this means that one can apply product operators in ⊠-terms in arbitrary order, and again *intuitively*, this means that the order in which protocol processes exchange messages at run-time does not matter. Nevertheless, to us, better understanding this intuition and making it formally precise is crucially important: it may give us clues on how to deal with the nonassociativity of ⊡, where our intuition fails us. Otherwise, we really have no choice but to resort to strict implementations of ⊡-terms, whose fixed communication order seriously hurts performance. At this point, we thus want to find a theoretical justification for why a *loose implementation* of a ⊠-term t (i.e., a distributed implementation of t in which protocol processes exchange messages more freely than in a strict implementation) is in fact a correct implementation of t.

Let Alice, Bob, and Carol be names for α, β, and γ in Fig. 2, and let q_α, q_β, and q_γ denote their states. First, we observe that considering associativity only at the syntactic level of ⊠-terms *at compile-time* is insufficient to make precise why the order in which Alice, Bob, and Carol exchange messages *at run-time* does not matter. To see this, note that although associativity implies that a strict implementation of $t_1 = (\alpha$ [x] $\beta)$ [x] γ is behaviorally equivalent to a strict implementation of $t_2 = \alpha$ [x] $(\beta$ [x] $\gamma)$, those strict implementations still statically fix the order in which protocol processes exchange messages. Instead, we should go one level deeper and consider how associativity influences the meanings of t_1 and t_2, shown in Fig. 10. At this level, the *inner structure of states* fixes the order in which Alice, Bob, and Carol exchange messages as part of their consensus algorithm (i.e., as part of the execution of the event-handling routine in Fig. 6b), to fire their composite {A , B , C , D}-transition. Note that states get this inner structure through the application of product; this structure is not something that we artificially impose out of nowhere (we merely ignored it so far). For instance, if we compose Alice, Bob, and Carol in that order, the state space of the resulting product automaton is $(Q_\alpha \times Q_\beta) \times Q_\gamma$, which consists of state $((q_\alpha\,,\, q_\beta)\,,\, q_\gamma)$ (whose parentheses convey its inner structure), whereas if we compose Bob, Carol, and Alice in that order, the state space of the resulting product automaton is $Q_\alpha \times (Q_\beta \times Q_\gamma)$, which consists of state $(q_\alpha\,,\, (q_\beta\,,\, q_\gamma))$ (whose parentheses convey that this state has a different inner structure than the previous state). The PA in Fig. 10 in fact differ from each other only in this respect. However, what both PA fail to capture—which is also why one can argue that a loose implementation does not correctly implement t_1 or t_2—is that a loose implementation of t_1 or t_2 essentially moves parentheses between substates around at run-time. Another way to look at this is that a loose implementation removes/ignores parentheses, as also common in notation for terms with associative and commutative operators. For instance, Alice, Bob, and Carol may initially communicate with each other in an order compliant with state $q_{(\alpha\beta)\gamma}$ (in Fig. 10a), but at some point, the loose implementation may evolve such that Alice, Bob, and Carol afterward communicate with each other in an order compliant with state $q_{\alpha(\beta\gamma)}$ (in Fig. 10b). Although such a loose implementation strictly speaking is neither a correct implementation of t_1 nor of t_2, we can use the meanings of t_1 and t_2 to construct a weakly bisimilar, "huge" PA that models the behavior

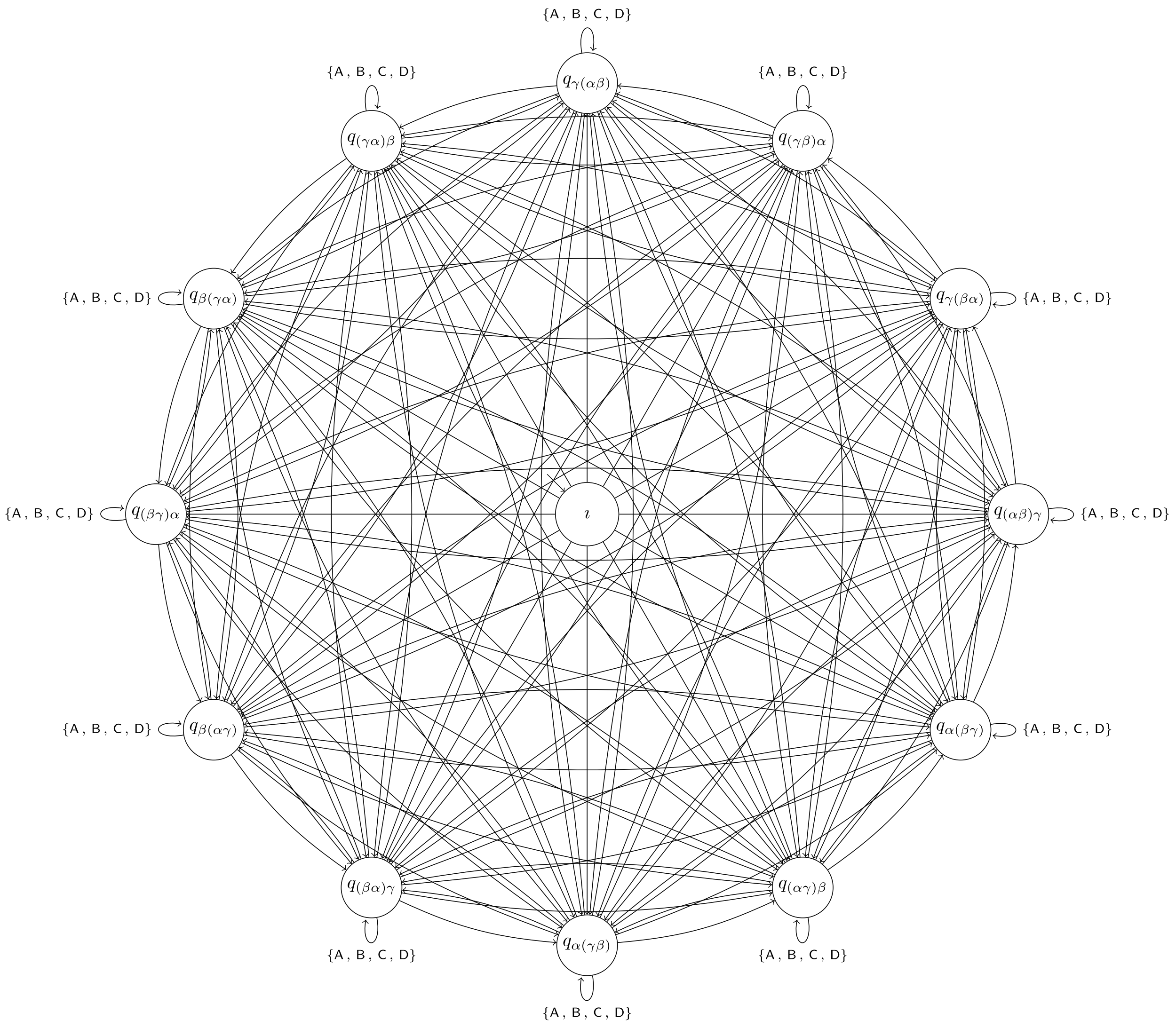
{A, B, C, D}
$q_{\gamma(\alpha\beta)}$
$q_{(\gamma\alpha)\beta}$
$q_{(\gamma\beta)\alpha}$
$q_{\beta(\gamma\alpha)}$
$q_{\gamma(\beta\alpha)}$
$q_{(\beta\gamma)\alpha}$
$\imath$
$q_{(\alpha\beta)\gamma}$
$q_{\beta(\alpha\gamma)}$
$q_{\alpha(\beta\gamma)}$
$q_{(\beta\alpha)\gamma}$
$q_{(\alpha\gamma)\beta}$
$q_{\alpha(\gamma\beta)}$

Fig. 11. Huge automaton for ⊠-term $(\alpha \texttt{ [x] } \beta) \texttt{ [x] } \gamma$. State $\imath$ in the center is the initial state. All unlabeled transitions represent silent transitions.

of the loose implementation. In particular, in this huge PA, shown in Fig. 11, we explicitly represent internal evolution steps as internal, silent transitions.

Generally, for an arbitrary ⊠-term t, the construction of its corresponding huge PA works as follows. Let $=_{\text{AC}} \subseteq \mathbb{P}\text{A} \times \mathbb{P}\text{A}$ denote the smallest congruence relation satisfying the following rules (i.e., add rules for reflexivity, symmetry, transitivity, and compositionality):

$$\frac{\alpha\,,\,\beta \in \mathbb{P}\text{A}}{\alpha \boxtimes \beta =_{\text{AC}} \beta \boxtimes \alpha} \qquad \frac{\alpha\,,\,\beta\,,\,\gamma \in \mathbb{P}\text{A}}{\alpha \boxtimes (\beta \boxtimes \gamma) =_{\text{AC}} (\alpha \boxtimes \beta) \boxtimes \gamma}$$

Essentially, $=_{\text{AC}}$ is equality up-to associativity and commutativity. Let $\mathbb{P}\text{A}/{=_{\text{AC}}}$ denote the quotient set of $\mathbb{P}\text{A}$ by $=_{\text{AC}}$, and let $A \in \mathbb{P}\text{A}/{=_{\text{AC}}}$ denote an equivalence class such that $[\![t]\!] \in A$. Note that because ⊠ is associative and commutative, all PA in A are pairwise bisimilar. To construct the huge PA for t, first, take the union of all PA in A (including $[\![t]\!]$). Second, add a new initial state to the constructed union, and connect this new initial state with a silent transition to the (former) initial states of the constituent PA from A. Finally, add a silent transition between all bisimilar states of constituent PA from A, in both directions. Formally, we have the following. Let $\mathcal{R}$ denote the largest set of bisimulation relations such that $R \in \mathcal{R}$ implies that $\alpha \approx^R \beta$ holds for some $\alpha\,,\,\beta \in A$.

$$
\begin{aligned}
Q &= \bigcup\{\mathsf{State}(\alpha) \mid \alpha \in A\} \cup \{\iota\} \\
P &= \bigcup\{\mathsf{Port}(\alpha) \mid \alpha \in A\} \\
&= \bigcap\{\mathsf{Port}(\alpha) \mid \alpha \in A\} \\
\longrightarrow &= \bigcup\{\mathsf{Trans}(\alpha) \mid \alpha \in A\} \cup \{(\iota\,,\emptyset,\top\,,\mathsf{init}(\alpha)) \mid \alpha \in A\} \cup \{(q\,,\emptyset\,,\top\,,q') \mid q\,R\,q' \ \textbf{and}\ R \in \mathcal{R}\} \\
\iota &= \text{a fresh state}
\end{aligned}
$$

A huge PA $\boldsymbol{\alpha}$ constructed for a $\boxtimes$-term t provides the theoretical justification of loose implementations: a loose implementation of t is in fact a correct implementation of t, because (i) $\boldsymbol{\alpha}$ models the execution of that loose implementation, including the evolution of the order in which protocol processes exchange messages, which $[\![t]\!]$ does not, and because (ii) $\boldsymbol{\alpha}$ is nevertheless weakly bisimilar to $[\![t]\!]$. This weak bisimilarity follows from the construction of $=_{\mathrm{AC}}$-equivalence class A, whose elements are pairwise bisimilar (because $\boxtimes$ is associative and commutative up-to bisimilarity). Note that in practice, a compiler never actually constructs the huge PA to generate code; we just use it here as means to formally reason about loose implementations.

The previous paragraphs, and in particular the definition of $=_{\mathrm{AC}}$, makes the importance of $\boxtimes$'s associativity precise: associativity (together with commutativity) characterizes a set of equivalent PA, each of which fixes a different order of exchanging messages, into which the loose implementation can evolve at run-time. Without associativity, such an equivalence class becomes much smaller, and the corresponding loose implementation, although a correct implementation, has limited freedom in evolving—it is less loose. Because performance decreases as strictness increases (i.e., as equivalence classes shrink), whenever associativity does not hold—as for $\boxdot$—we should strive to make equivalence classes as broad as possible by using other equivalence properties. Otherwise, we can only resort either to inefficient strict implementations or to efficient loose implementations whose correctness we have not formally established.

5.2. Hybrid approach

As in distributed implementations, in partially-distributed/partially-centralized hybrid implementations of Reo connectors, we have a number of protocol processes, each of which executes a variant of the event-handling routine in Fig. 6b. Ideally, as in distributed implementations, we do not restrict the order in which protocol processes execute this routine and, as part of it, exchange messages with each other. However, every hybrid implementation corresponds to a nonassociative $\boxdot$-term instead of to an associative $\boxtimes$-term. Consequently, we cannot use exactly the same technique as in the previous subsection, which relies on associativity in the definition of $=_{\mathrm{AC}}$, for establishing that a restriction-free, loose implementation of a $\boxdot$-term is in fact a correct implementation. However, we *can* use a similar approach to obtain a somewhat weaker but, perhaps surprisingly, equally usable result: using a different equivalence relation instead of $=_{\mathrm{AC}}$, we can construct huge PA for modeling the execution of hybrid implementations that exhibit some acceptable degree of looseness, stricter than before but still reasonably loose. To do this, we first investigate under which circumstances one can move parentheses and PA around in $\boxdot$-terms, which requires nontrivial technical machinery. Afterward, we proceed with a construction of huge PA for $\boxdot$-terms and relate those huge PA to communication orders in hybrid implementations.

5.2.1. Moving parentheses

First, we study under which circumstances explicitly “inserting” parentheses in a $\boxdot$-term $\alpha_1 \,[\,\cdot\,]\, \cdots \,[\,\cdot\,]\, \alpha_m$ is semantics-preserving (with respect to implicit insertion of parentheses according to the right-associative notation that we have assumed so far). As a first step, we investigate this for $m = 3$, in which case we get a form of conditional associativity: the following lemma states that if Alice is a conditional slave of Bob, and both Alice and Bob are independent of Carol, associativity applies.

Lemma 12. $\big[\alpha \Mapsto \beta$ **and** $\alpha\,,\,\beta \asymp \gamma\big]$ **implies** $\alpha \boxdot (\beta \boxdot \gamma) \approx (\alpha \boxdot \beta) \boxdot \gamma$.

Proof (Outline). By applying Definition 10 of $\Mapsto$ to the premise of the lemma, we can derive:

$$\alpha \Mapsto \beta\,,\,\gamma \ \textbf{and}\ \beta \Mapsto \gamma$$

Then, by applying Lemmas 11 and 10 and Theorem 1, we can derive $\alpha \boxdot (\beta \boxdot \gamma) = \alpha \boxtimes (\beta \boxtimes \gamma)$. Then, because $\boxtimes$ is associative up-to bisimilarity, we can derive $\alpha \boxdot (\beta \boxdot \gamma) \approx (\alpha \boxtimes \beta) \boxtimes \gamma$. The lemma subsequently follows by similarly applying Lemmas 11 and 10 and Theorem 1 (and because $\boxtimes$ is commutative up-to bisimilarity). See [27, Appendix C, Lemma 13] for a detailed proof. □

We lift this result to arbitrary PA in $\mathcal{B}$. The following lemma states that for all PA $\alpha_1 \boxdot \cdots \boxdot \alpha_k \boxdot \beta_1 \boxdot \cdots \boxdot \beta_l \in \mathcal{B}$, inserting parentheses around two successive betas, β_{j-1} and β_j, is semantics-preserving. Moreover, $\mathcal{B}$ is closed under such parentheses insertion. Because all lemmas in this subsection hold for PA in $\mathcal{B}$, closedness ensures that those lemmas are applicable not only before but also after inserting parentheses.

Lemma 13. $\big[1 < j \leq l$ **and** $[\alpha]_1^k[\beta]_1^l \in \mathcal{B}\big]$ **implies** $\begin{bmatrix} [\alpha]_1^k[\beta]_1^{j-2}(\beta_{j-1}\beta_j)[\beta]_{j+1}^l \in \mathcal{B} \textbf{ and} \\ [\alpha]_1^k[\beta]_1^{j-2}(\beta_{j-1}\beta_j)[\beta]_{j+1}^l \approx [\alpha]_1^k[\beta]_1^l \end{bmatrix}$.

Proof (Outline). To prove closedness, by Definition 12 of $\mathcal{B}$, we must prove that every alpha is a conditional slave of $\beta_{j-1}\beta_j$ and that every beta is independent of $\beta_{j-1}\beta_j$. To show the former, by applying Definition 12 of $\mathcal{B}$ to the premise of the lemma, we can derive $\big[\alpha_i \mapsto\!\!\!\Rightarrow \beta_{j_2}$ **for all** $1 \leq i \leq k\,,\ 1 \leq j_2 \leq l\big]$. Then, by pulling out $j_2 = j-1$ and $j_2 = j$ and by applying Lemma 11, we can derive:

$$\alpha_i \mapsto\!\!\!\Rightarrow \beta_{j-1}\beta_j \textbf{ for all } 1 \leq i \leq k$$

To show the latter, by applying Definition 12 of $\mathcal{B}$ to the premise of the lemma, we can derive:

$$\big[j_1 \neq j_2 \textbf{ implies } \beta_{j_1} \asymp \beta_{j_2}\big] \textbf{ for all } 1 \leq j_1\,,\ j_2 \leq l$$

Then, by pulling out $j_2 = j-1$ and $j_2 = j$ and by applying Definition 8 of $\asymp$, basic set theory, Proposition 1, and again Definition 8, we can derive $\big[[j_2 \notin \{j-1\,,\ j\}$ **implies** $\beta_{j-1}\beta_j \asymp \beta_{j_2}\big]$ **for all** $1 \leq j_2 \leq l\big]$. Closedness subsequently follows from combining these two results.

To prove equivalence, by applying Definition 12 of $\mathcal{B}$ to the premise of the lemma, we can derive:

$$\big[j_1 \neq j_2 \textbf{ implies } \beta_{j_1} \asymp \beta_{j_2}\big] \textbf{ for all } 1 \leq j_1\,,\ j_2 \leq l$$

Then, by pulling out $j_1 = j-1$ and $j_2 = j$, we can derive:

$$\beta_{j-1} \asymp \beta_j \textbf{ and } \beta_{j-1} \asymp \beta_{j+1}\,,\ \ldots\,,\ \beta_l \textbf{ and } \beta_j \asymp \beta_{j+1}\,,\ \ldots\,,\ \beta_l$$

Then, by applying Corollary 1, Definition 10 of $\mapsto\!\!\!\Rightarrow$, Lemma 12, we can derive $\beta_{j-1}\beta_j[\beta]_{j+1}^l \approx (\beta_{j-1}\beta_j)[\beta]_{j+1}^l$. Equivalence subsequently follows from Lemma 2.

See [27, Appendix C, Lemma 16] for a detailed proof. □

The following corollary generalizes the previous lemma from two PA to a sequence of PA: it states that inserting parentheses around the first beta β_1 and the j-th beta β_j is semantics-preserving.

Corollary 3. $\big[1 \leq j \leq l$ **and** $[\alpha]_1^k[\beta]_1^l \in \mathcal{B}\big]$ **implies** $\begin{bmatrix} [\alpha]_1^k([\beta]_1^j)[\beta]_{j+1}^l \in \mathcal{B} \textbf{ and} \\ [\alpha]_1^k([\beta]_1^j)[\beta]_{j+1}^l \approx [\alpha]_1^k[\beta]_1^l \end{bmatrix}$.

Similar to Lemma 13, the following lemma states that for all PA in $\alpha_1 \boxdot \cdots \boxdot \alpha_k \boxdot \beta_1 \boxdot \cdots \boxdot \beta_l \in \mathcal{B}$, inserting parentheses around the last alpha α_k and the first beta β_1 is semantics-preserving if α_k is independent of all betas. Moreover, $\mathcal{B}$ is closed under this kind of parentheses insertion.

Lemma 14. $\big[[\alpha]_1^k[\beta]_1^l \in \mathcal{B}$ **and** $\alpha_k \asymp \beta_2\,,\ \ldots\,,\ \beta_l\big]$ **implies** $\begin{bmatrix} [\alpha]_1^{k-1}(\alpha_k\beta_1)[\beta]_2^l \in \mathcal{B} \textbf{ and} \\ [\alpha]_1^{k-1}(\alpha_k\beta_1)[\beta]_2^l \approx [\alpha]_1^k[\beta]_1^l \end{bmatrix}$.

Proof. Structurally, the proof is similar to the proof of Lemma 13. The main difference is that the premise of the lemma explicitly asserts $\alpha_k \asymp \beta_2\,,\ \ldots\,,\ \beta_l$. This is necessary because Definition 12 of $\mathcal{B}$ does not guarantee that α_k, unlike β_{j-1} in Lemma 13, is independent of all betas; it guarantees only that α_k is a conditional slave. See [27, Appendix C, Lemma 17] for a detailed proof. □

The following corollary generalizes the previous lemma from two PA to a sequence of PA: it states that one can insert parentheses around the i-th alpha α_i and the first beta β_1 in a semantics preserving way.

Corollary 4. $\Big[i \leq k$ **and** $[\alpha]_1^k[\beta]_1^l \in \mathcal{B}$ **and** $\begin{bmatrix} \alpha_{i'} \asymp \beta_2\,,\ \ldots\,,\ \beta_l \\ \textbf{for all} \\ k-i+1 \leq i' \leq k \end{bmatrix}\Big]$ **implies** $\begin{bmatrix} [\alpha]_1^{k-i}([\alpha]_{k-i+1}^k\beta_1)[\beta]_2^l \in \mathcal{B} \textbf{ and} \\ [\alpha]_1^{k-i}([\alpha]_{k-i+1}^k\beta_1)[\beta]_2^l \approx [\alpha]_1^k[\beta]_1^l \end{bmatrix}$.

5.2.2. Moving PA

Next, we study under which circumstances "swapping" PA in a $\boxdot$-term $\alpha_1\ [\,\cdot\,]\ \cdots\ [\,\cdot\,]\ \alpha_m$ is semantics-preserving. As before, we first investigate this for $m = 3$. The following two lemmas state two alternative conditions under which one can "swap" Alice and Bob. The first condition states that Alice and Bob are both each other's conditional slave and Carol's. The second condition states that Alice, Bob, and Carol are pairwise independent.

Lemma 15. $\big[\alpha \mapsto\!\!\!\Rightarrow \beta\,,\ \gamma$ **and** $\beta \mapsto\!\!\!\Rightarrow \alpha\,,\ \gamma\big]$ **implies** $\alpha \boxdot (\beta \boxdot \gamma) \approx \beta \boxdot (\alpha \boxdot \gamma)$.

Proof (Outline). Structurally, the proof is similar to the proof of Lemma 12. The main difference is that after establishing $\alpha \boxdot (\beta \boxdot \gamma) \approx (\alpha \boxtimes \beta) \boxtimes \gamma$, we now conclude $\alpha \boxdot (\beta \boxdot \gamma) \approx \beta \boxtimes (\alpha \boxtimes \gamma)$ (because $\boxtimes$ is commutative and associative up-to bisimilarity). The rest of the proof is similar. See [27, Appendix C, Lemma 14] for a detailed proof. □

Lemma 16. $\left[\alpha \asymp \beta\,,\ \gamma\ \textbf{ and }\ \beta \asymp \gamma\right]$ **implies** $\alpha \boxdot (\beta \boxdot \gamma) \approx \beta \boxdot (\alpha \boxdot \gamma)$.

Proof (Outline). Because $\asymp$ is symmetric, we can derive $\left[\alpha \asymp \beta\,,\ \gamma\ \textbf{ and }\ \beta \asymp \alpha\,,\ \gamma\right]$ from the premise of the lemma. Then, by applying Definition 10 of $\Mapsto$, we can derive $\left[\alpha \Mapsto \beta\,,\ \gamma\ \textbf{ and }\ \beta \Mapsto \alpha\,,\ \gamma\right]$. The lemma subsequently follows from Lemma 15. See [27, Appendix C, Lemma 15] for a detailed proof. □

To lift the previous results to arbitrary PA in $\mathcal{B}$, we first introduce four functions for moving PA left and right in a $\boxdot$-term. The two simpler ones are the *move-right* and *move-left* functions. The move-left function takes a PA $\alpha_1 \boxdot \cdots \boxdot \alpha_k \boxdot \beta_1 \boxdot \cdots \boxdot \beta_l \in \mathcal{B}$ and a constituent alpha α_i as input and recursively moves α_i rightward until it has become the rightmost alpha; the move-left function works similarly but for a beta and in the leftward direction.

Definition 13 *(Move-right/left functions).* The move-right and move-left functions, denoted by $\Rightarrow$ and $\Leftarrow$, are the functions from $\mathbb{P}\text{A} \times \mathcal{B}$ to $\mathbb{P}\text{A}$ defined by the following equation:

$$\Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) = \begin{cases} \Rightarrow\!(\alpha_i\,,\ [\alpha]_1^{i-1}\alpha_{i+1}\alpha_i[\alpha]_{i+2}^k[\beta]_1^l) & \textbf{if } 1 \leq i < k \\ [\alpha]_1^k[\beta]_1^l & \textbf{otherwise} \end{cases}$$

$$\Leftarrow\!(\beta_j\,,\ [\alpha]_1^k[\beta]_1^l) = \begin{cases} \Leftarrow\!(\beta_j\,,\ [\alpha]_1^k[\beta]_1^{j-2}\beta_j\beta_{j-1}[\beta]_{j+1}^l) & \textbf{if } 1 < j \leq l \\ [\alpha]_1^k[\beta]_1^l & \textbf{otherwise} \end{cases}$$

The following two functions generalize the move-right/left functions from single constituent PA to sets of constituent PA. To define these functions, we introduce the following notation. If $X = \{x_1\,,\ \ldots\,,\ x_n\}$ is a set and $X' \subseteq X$ is a subset of size $m \leq n$, we write $X'(i)$ for $1 \leq i \leq m$ to denote the i-th element of X' according to the indices associated with elements in X. For instance, if $X = \{x_1\,,\ x_2\,,\ x_3\,,\ x_4\,,\ x_5\}$ and $X' = \{x_2\,,\ x_4\,,\ x_5\}$, we have $X'(1) = x_2$ and $X'(2) = x_4$ and $X'(3) = x_5$.

Definition 14 *(Move-all-right/left functions).* The move-all-right and move-all-left functions, denoted by $\Rrightarrow$ and $\Lleftarrow$, are the functions from $\wp(\mathbb{P}\text{A}) \times \mathcal{B}$ to $\mathbb{P}\text{A}$ defined by the following equation:

$$\Rrightarrow\!(A\,,\ [\alpha]_1^k[\beta]_1^l) = \begin{cases} \Rrightarrow\!(A \setminus \{A(1)\}\,,\ \Rightarrow\!(A(1)\,,\ [\alpha]_1^k[\beta]_1^l)) & \textbf{if } A \subseteq \{\alpha_1\,,\ \ldots\,,\ \alpha_k\} \\ [\alpha]_1^k[\beta]_1^l & \textbf{otherwise} \end{cases}$$

$$\Lleftarrow\!(B\,,\ [\alpha]_1^k[\beta]_1^l) = \begin{cases} \Lleftarrow\!(B \setminus \{B(1)\}\,,\ \Leftarrow\!(B(1)\,,\ [\alpha]_1^k[\beta]_1^l)) & \textbf{if } B \subseteq \{\beta_1\,,\ \ldots\,,\ \beta_l\} \\ [\alpha]_1^k[\beta]_1^l & \textbf{otherwise} \end{cases}$$

The following lemma states that the move-right function indeed has the effect of moving a PA α_i rightward (effectiveness), that $\mathcal{B}$ is closed under such moving (closedness), and that such moving is semantics-preserving (equivalence). Afterward, another lemma states the same properties for moving a PA β_j leftward.

Lemma 17. $\left[1 \leq i \leq k\ \textbf{ and }\ [\alpha]_1^k[\beta]_1^l \in \mathcal{B}\right]$ **implies** $\left[\begin{array}{l} \Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) = [\alpha]_1^{i-1}[\alpha]_{i+1}^k\alpha_i[\beta]_1^l \\ \textbf{and } \Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) \in \mathcal{B} \\ \textbf{and } \Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) \approx [\alpha]_1^k[\beta]_1^l \end{array}\right]$.

Proof (Outline). We prove the lemma by reverse induction on i. The base case, $i = k$, follows straightforwardly. In the inductive step, $1 \leq i < k$, we assume that the lemma holds for all $i < i' \leq k$ as our induction hypothesis. First, we can construct a PA $[\hat{\alpha}]_1^k[\beta]_1^l$ such that $\hat{\alpha}_{\hat{\imath}} = \alpha_i$ for all $\hat{\imath} \notin \{i\,,\ i+1\}$ and $\hat{\alpha}_i = \alpha_{\hat{\imath}+1}$ and $\hat{\alpha}_{\hat{\imath}+1} = \alpha_i$. This PA is in $\mathcal{B}$, because by Definition 12, membership of $\mathcal{B}$ is invariant under the order of the alphas. Then, by applying the induction hypothesis to $[\hat{\alpha}]_1^k[\beta]_1^l$ for $i' = i + 1$, we can derive:

$$\Rightarrow\!(\hat{\alpha}_{i'}\,,\ [\hat{\alpha}]_1^k[\beta]_1^l) = [\hat{\alpha}]_1^{i'-1}[\hat{\alpha}]_{i'+1}^k\hat{\alpha}_{i'}[\beta]_1^l\ \textbf{ and }\ \Rightarrow\!(\hat{\alpha}_{i'}\,,\ [\hat{\alpha}]_1^k[\beta]_1^l) \in \mathcal{B}\ \textbf{ and }\ \Rightarrow\!(\hat{\alpha}_{i'}\,,\ [\hat{\alpha}]_1^k[\beta]_1^l) \approx [\hat{\alpha}]_1^k[\beta]_1^l$$

To prove effectiveness, by applying Definition 13 of $\Rightarrow$ and the definition of the hat-alphas to $[\alpha]_1^k[\beta]_1^l$, we can derive $\Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) = \Rightarrow\!(\hat{\alpha}_{i'}\,,\ [\hat{\alpha}]_1^k[\beta]_1^l)$. Then, by substituting the left consequence of the induction hypothesis above, we can derive $\Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) = [\hat{\alpha}]_1^{i'-1}[\hat{\alpha}]_{i'+1}^k\hat{\alpha}_{i'}[\beta]_1^l$. Then, by applying the definition of the hat-alphas, we can derive $\Rightarrow\!(\alpha_i\,,\ [\alpha]_1^k[\beta]_1^l) = [\alpha]_1^{i-1}[\alpha]_{i+1}^k\alpha_i[\beta]_1^l$.

To prove closedness, by applying the definition of the hat-alphas and Definition 13 of $\Rightarrow$ to the middle consequence of the induction hypothesis above, we can derive $\Rightarrow(\alpha_i\,,\,[\alpha]_1^k[\beta]_1^l) \in \mathcal{B}$.

To prove equivalence, by applying Definition 13 of $\Rightarrow$ and the definition of the hat-alphas to $[\alpha]_1^k[\beta]_1^l$, we can derive $\Rightarrow(\alpha_i\,,\,[\alpha]_1^k[\beta]_1^l) = \Rightarrow(\hat{\alpha}_{i'}\,,\,[\hat{\alpha}]_1^k[\beta]_1^l)$. Then, by applying the right consequence of the induction hypothesis above, we can derive $\Rightarrow(\alpha_i\,,\,[\alpha]_1^k[\beta]_1^l) \approx [\hat{\alpha}]_1^k[\beta]_1^l$. Then, by applying the definition of the hat-alphas, we can derive $\Rightarrow(\alpha_i\,,\,[\alpha]_1^k[\beta]_1^l) \approx [\alpha]_1^{i-1}\alpha_{i+1}\alpha_i[\alpha]_{i+2}^k[\beta]_1^l$. Then, by applying Definition 12 of $\mathcal{B}$, Corollary 2, and Lemma 15, we can derive $\Rightarrow(\alpha_i\,,\,[\alpha]_1^k[\beta]_1^l) \approx [\alpha]_1^k[\beta]_1^l$.

See [27, Appendix C, Lemma 18] for a detailed proof. □

Lemma 18. $\left[1 \leq j \leq l \textbf{ and } [\alpha]_1^k[\beta]_1^l \in \mathcal{B}\right]$ **implies** $\begin{bmatrix} \Leftarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) = [\alpha]_1^k\beta_j[\beta]_1^{j-1}[\beta]_{j+1}^l \\ \textbf{and } \Leftarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) \in \mathcal{B} \\ \textbf{and } \Leftarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) \approx [\alpha]_1^k[\beta]_1^l \end{bmatrix}$.

Proof. Structurally, the proof is similar to the proof of Lemma 17. The main difference is the use of Corollary 1 (instead of Corollary 2) and Lemma 16 (instead of Lemma 15) in the equivalence proof. See [27, Appendix C, Lemma 19] for a detailed proof. □

5.2.3. Reordering masters and slaves

Finally, we define a *reorder function* that takes a PA $\alpha_1 \boxdot \cdots \boxdot \alpha_k \boxdot \beta_1 \boxdot \cdots \boxdot \beta_l \in \mathcal{B}$ and a constituent beta β_j as input and (i) moves β_j leftward, (ii) moves all slaves A of β_j rightward, (iii) moves all other masters B of the slaves A, beside β_j, leftward, and (iv) inserts parentheses around A, B, and β_j. Let $\mathsf{Slave}(\beta\,,\,A)$ denote the set of all slaves of PA β in a set of PA A. Similarly, let $\mathsf{Master}(\alpha\,,\,B)$ denote the set of all masters of α in a set of PA B. (In the following definition, step (ii) and step (iii) actually occur in the reverse order: we first move masters B rightward and only afterward move slaves A leftward.)

Definition 15 *(Reorder function)*. The reorder function, denoted by $\Leftrightarrow$, is the function from $\mathbb{P}\mathrm{A} \times \mathcal{B}$ to $\mathbb{P}\mathrm{A}$ defined by the following equation:

$$\Leftrightarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) = \begin{cases} [\tilde{\alpha}]_1^{k-|A|}\left([\tilde{\alpha}]_{k-|A|+1}^k[\tilde{\beta}]_1^{|B|+1}\right)[\tilde{\beta}]_{|B|+2}^l & \textbf{if } 1 \leq j \leq l \\ [\alpha]_1^k[\beta]_1^l & \textbf{otherwise} \end{cases}$$

$$\textbf{for} \begin{bmatrix} A = \mathsf{Slave}(\beta_j\,,\,\{\alpha_1\,,\,\ldots\,,\,\alpha_k\}) \\ B = (\bigcup_{\alpha \in A} \mathsf{Master}(\alpha, \{\beta_1\,,\,\ldots\,,\,\beta_l\})) \setminus \{\beta_j\} \\ [\tilde{\alpha}]_1^k[\tilde{\beta}]_1^l = \Rightarrow(A\,,\,\Leftarrow(B\,,\,\Leftarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l))) \end{bmatrix}$$

The following theorem states that $\mathcal{B}$ is closed under reordering and that reordering is semantics-preserving.

Theorem 3. $\left[1 \leq j \leq l \textbf{ and } [\alpha]_1^k[\beta]_1^l \in \mathcal{B}\right]$ **implies** $\begin{bmatrix} \Leftrightarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) \in \mathcal{B} \textbf{ and} \\ \Leftrightarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) \approx [\alpha]_1^k[\beta]_1^l \end{bmatrix}$.

Proof (Outline). Let A, B, and $[\tilde{\alpha}]_1^k[\tilde{\beta}]_1^l$ be defined as in Definition 15 of $\Leftrightarrow$.

First, by applying the definition of Master, Definition 12 of $\mathcal{B}$, Definition 10 of $\bowtie$, and basic set theory, we can derive that all α in A and β outside $B \cup \{\beta_j\}$ are independent. Then, by applying set theory, we can derive $\left[\tilde{\alpha}_{i'} \asymp \tilde{\beta}_{|B|+2}\,,\,\ldots\,,\,\tilde{\beta}_l \textbf{ for all } k-|A|+1 \leq i' \leq k\right]$. In words: every tilde-alpha indexed higher than $k-|A|$ (i.e., every alpha in A, moved rightward in $[\tilde{\alpha}]_1^k[\tilde{\beta}]_1^l$) is independent of every tilde-beta indexed higher than $|B|+1$ (i.e., every beta in B, moved leftward in $[\tilde{\alpha}]_1^k[\tilde{\beta}]_1^l$).

Second, by applying Lemma 17 (for β_j and for every beta in B), Lemma 18 (for every alpha in α), and basic set theory to $[\alpha]_1^k[\beta]_1^l$, we can derive $[\tilde{\alpha}]_1^k[\tilde{\beta}]_1^l \in \mathcal{B}$. Then, by applying Corollary 3, we can derive $\left[[\tilde{\alpha}]_1^k([\tilde{\beta}]_1^{|B|+1})[\tilde{\beta}]_{|B|+2}^l \in \mathcal{B} \textbf{ and } [\tilde{\alpha}]_1^k([\tilde{\beta}]_1^{|B|+1})[\tilde{\beta}]_{|B|+2}^l \approx [\alpha]_1^k[\beta]_1^l\right]$.

After combining the previous two results, by applying Corollary 4, we can derive:

$$[\tilde{\alpha}]_1^{k-|A|}([\tilde{\alpha}]_{k-|A|+1}^k[\tilde{\beta}]_1^{|B|+1})[\tilde{\beta}]_{|B|+2}^l \in \mathcal{B} \textbf{ and } [\tilde{\alpha}]_1^{k-|A|}([\tilde{\alpha}]_{k-|A|+1}^k[\tilde{\beta}]_1^{|B|+1})[\tilde{\beta}]_{|B|+2}^l \approx [\tilde{\alpha}]_1^k[\tilde{\beta}]_1^l$$

The theorem subsequently follows from Definition 15.

See [27, Appendix C, Theorem 3] for a detailed proof. □

With the reorder function, we now define a construction of huge PA for $\boxdot$-terms, intended to model the execution of hybrid implementations. Let $=_{\Leftrightarrow} \subseteq \mathbb{P}\mathrm{A} \times \mathbb{P}\mathrm{A}$ denote the smallest congruence relation satisfying the following rules (i.e., add rules for reflexivity, symmetry, transitivity, and compositionality):

$$\frac{\alpha\,,\,\beta \in \mathbb{P}\text{A}}{\alpha \boxdot \beta =_{\Leftrightarrow} \beta \boxdot \alpha} \qquad \frac{\begin{array}{l} \left[\Leftrightarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l) = [\alpha']_1^k[\beta']_1^l \ \textbf{ for some } j\right] \\ \textbf{or } \left[\Leftrightarrow(\beta'_j\,,\,[\alpha']_1^k[\beta']_1^l) = [\alpha]_1^k[\beta]_1^l \ \textbf{ for some } j\right] \end{array}}{[\alpha]_1^k[\beta]_1^l =_{\Leftrightarrow} [\alpha']_1^k[\beta']_1^l}$$

Now, to construct huge PA for hybrid implementations, we perform the same steps as before but take the quotient of $\mathbb{P}$A by $=_{\Leftrightarrow}$ instead of by $=_{\text{AC}}$. Observe that as before, we construct this huge PA only for our analysis; the compiler never actually constructs any such automaton. Also, note that because $\boxdot$ is commutative up-to bisimilarity and by Theorem 3, all PA in every equivalence class $A \in \mathbb{P}\text{A}/{=_{\Leftrightarrow}}$ are pairwise bisimilar, just as for equivalence classes from $\mathbb{P}\text{A}/{=_{\text{AC}}}$. As before, this ensures that the constructed huge PA for A is weakly bisimilar to every PA in A.

What remains is relating huge PA constructed in this way to actual hybrid implementations. Essentially, a silent transition from a state in $[\alpha]_1^k[\beta]_1^l$ to a bisimilar state in $\Leftrightarrow(\beta_j\,,\,[\alpha]_1^k[\beta]_1^l)$ models an evolution step of a loose hybrid implementation after which the protocol process for β_j can start communicating with a number of other masters (collected in B in Definition 15 of $\Leftrightarrow$) and with a number of slaves (collected in A). To see this, note that after the evolution step, β_j is (one of) the most deeply nested PA. Because one can apply $\Leftrightarrow$ to any beta in $[\alpha]_1^k[\beta]_1^l$, every beta (i.e., every master) can assume the role of β_j. This means that a loose hybrid implementation for a $\boxdot$-term t is in fact a correct implementation of t if this implementation ensures that only betas (i.e., masters) start exchanging messages: theoretically, such an implementation can always fire a silent transition to reach a state in which a master that starts communicating is most deeply nested. In contrast, alphas (i.e., slaves) may never start communicating.

Obviously, loose hybrid implementations (i.e., those modeled by $=_{\Leftrightarrow}$-based huge PA) are in theory stricter about communication order than loose distributed implementations (i.e., those modeled by $=_{\text{AC}}$-based huge PA, which impose no restrictions at all), but in practice, the extra restrictions in hybrid implementations do not matter. The reason is that slaves correspond to the asynchronous regions of a connector, whose protocol processes never share ports with computation processes: the protocol process for every asynchronous region is separated from every computation process by at least one protocol process for a synchronous regions (in the simplest case: one for a basic synchronous channel). Consequently, a protocol process for an asynchronous region never has to handle events originating from a computation process (i.e., I/O-operations) and therefore never starts exchanging messages anyway. Thus, the restriction that we found through our previous formal analysis is a necessary prerequisite for the correctness of loose hybrid implementations, does not actually impose any real restrictions in practice, and therefore does not hurt performance.

6. Related work

Reo Closest to ours is the work on splitting connectors into (a)synchronous regions for better performance. Proença developed the first implementation based on these ideas, demonstrated its merit through benchmarks, and invented an automaton model—*behavioral automata*—to reason about split connectors in his PhD thesis and associated publications [11–13]. Furthermore, Clarke and Proença explored connector splitting in the context of the connector coloring semantics [10]. 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 version. To address this problem, Clarke and Proença propose a new variant—*partial connector coloring*—which allows one to better model locality and independencies between different parts of a connector. Recently, Jongmans et al. studied a formal justification of connector splitting in a process algebraic setting, without data [29], and with data [30]. Although, as shown in Section 4.2, one can use the notion of (a)synchronous regions to apply our results to code generation for connectors, our results go beyond that. (They can, for instance, also be applied to code generation for Web service proxies in Reo-based orchestrations [5,6].)

Also related to the work presented in this paper is the work of Kokash et al. on *action constraint automata* (ACA) [25]. Kokash et al. argue that ordinary port/constraint automata describe the behavior of Reo connectors too coarsely, which makes it impossible to express certain fine parallel behavior. In contrast, ACA have more flexible transition labels which, for instance, allow one to explicitly model the start and end of interaction on a particular port (one cannot make this distinction using port/constraint automata). Consequently, ACA better describe the behavior of existing connector implementations (under certain assumptions). However, the increased granularity of ACA comes at the price of substantially larger models. This makes them less suitable for code generation.

In this paper, we developed a middle ground between the distributed approach and the centralized approach to implementing Reo connectors: we took the distributed approach as our starting point and proposed to apply $\boxtimes$ to small automata at compile-time to obtain medium automata for (a)synchronous regions. Alternatively, one could start with the centralized approach and try to *decompose* a big automaton into medium automata. Because decomposing a big automaton seems much harder than composing small automata, we opted for the former strategy. Others have studied decomposition of Reo connectors, albeit in different contexts. For instance, Koehler and Clarke investigated decomposition of PA [19]. They showed that one can decompose every PA into instances of only two PA. Essentially, this means that one can construct every Reo connector expressible in terms of PA from instances of only two different primitive connectors. Pourvatan et al. explored the decomposition of *complete constraint automata* [31], an extension of constraint automata. Their approach differs significantly from the work of Koehler and Clarke: Pourvatan et al. develop an *inverse product*, which allows them to factor out certain parts of a complete constraint 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. However, suppose that we have no specification of the connector that should coordinate those components. The technique of Pourvatan et al. now allows us to factor out the component automata from the system automaton to get the automaton for the connector. Pourvatan et al. exemplify this with a service-oriented application.

Distributed coordination We proceed with related work on distributed coordination.

In [32], Gelernter introduces Linda, historically the first genuine member of the *data-driven* family of coordination languages [33]. Central to Linda is the concept of a *tuple space* in which both computation processes and tuples of data, originating from and accessible to those processes, float. Although a tuple space gives the programmer the illusion of shared memory, at the hardware level, memory may actually be distributed over n locations. Several approaches to implementing physically distributed tuple spaces exist. For instance, one can maintain the entire tuple space at one of the n locations (e.g., Feng et al. [34], Wyckoff et al. [35]), but although simple to implement, this does not scale well in the number of computation processes [36]. The centralized approach to implementing Reo connectors has a similar scalability problem (due to oversequentialization). Alternatively, one can scatter (with or without replication) the tuples in the tuple space over all n locations. Although such an approach has better scalability, one must resolve several issues to obtain a working implementation, such as deciding where to store which tuple, efficiently retrieving tuples, and load balancing [33]. Examples include the work by Bjornson [37], Feng et al. [36], Rowstron & Wood [38], Menezes & Tolksdorf [39], and Atkinson [40]. In contrast to languages based on tuple spaces, Reo belongs to the *control-driven* family of coordination languages [33]. Distributed tuple space implementations therefore differ fundamentally from distributed/hybrid Reo implementations as discussed in this paper: distribution of data (tuples) versus distribution of control (small/medium automata).

In [41], Bonakdarpour et al. present an approach for generating distributed implementations for specifications in BIP [42], a framework for specifying component-based systems at three specification levels: behavior of components, interaction between components, and priorities on interactions. BIP forbids simultaneous execution of conflicting interactions (i.e., interactions that require the same resource), and a key aspect discussed by Bonakdarpour et al. is ensuring that such conflicting interactions execute mutually exclusively in distributed implementations of BIP specifications. For this, Bonakdarpour et al. propose a three-layered implementation architecture: the bottom layer consists of distributed components, the middle layer consists of a number of interaction execution engines, each responsible for executing its own subset of all interactions, and a top layer for resolving potential conflicts. Compared to our work, a set of BIP interactions roughly coincides with the transitions of a (single-state) PA, as recently formally shown by Dokter et al. [43], and the middle layer of execution engines roughly coincides with our $\boxdot$-terms of "medium" PA (i.e., the regions of a connector; see Section 4.2). One difference is that we do not consider a bottom layer of distributed components (because Reo is oblivious to the entities under coordination). A more important difference is that Bonakdarpour et al. aim for a finer distribution granularity than we do, which requires them to handle conflicting interactions with their third layer. We avoid this problem by amalgamating PA with "conflicting transitions" with each other to form one medium PA, effectively serializing those transitions at run-time. In our setting, for performance reasons, we prefer firing such transitions sequentially over adding an algorithm for conflict resolution.

Nonassociativity Closest to our work on compensating for $\boxdot$'s nonassociativity seem references to nonassociative parallel composition operators in the literature on concurrency theory, where such operators are usually considered defective. For instance, in [44], Vrancken extends the Algebra of Communicating Processes [45] (ACP) with the empty process ε (i.e., the neutral element for sequential composition). In particular, Vrancken improves an earlier extension of ACP with ε in which ACP's merge operator "unfortunately [...] turned out not associative" [44, page 291]. Baeten and Van Glabbeek similarly state that "this problem [a nonassociative merge operator] was remedied" [46, page 153] by Vrancken. In the context of timed automata with shared variables and action synchronization (as in UPPAAL [47]), Berendsen & Vaandrager point out that "the approach [to support shared variables and action synchronization] in [6] [sic] is flawed since parallel composition is not associative" [48, page 234]. Later, they state that "commutativity and associativity are highly desirable properties for parallel composition operators" [48, page 240]. In [49], Anantharaman et al. consider a process algebra with a nonassociative synchronous composition operator. However, Anantharaman et al. subsequently characterize a class of processes for which this operator actually is associative and work only with processes from that class to model systems. In [50], Segala discusses problems of defining a parallel composition operator for general probabilistic automata, symptomized by nonassociativity. Finally, in [51], Klin & Sassone notice that parallel composition in the stochastic π-calculus [52] is generally nonassociative and investigate under which conditions it is.

However, nonassociativity seems not always problematic. For instance, in [53], in the context of reachability analysis, Yeh investigates a state space reduction technique for processes by adding distinguished actions for suspending and resuming processes; in the resulting theory, parallel composition is nonassociative. In [54], Kuske & Meinecke introduce a nonassociative product operator on branching automata [55] with costs. Finally, the Orc orchestration language has three combinators to express parallel execution, two of which are nonassociative (and noncommutative) by design, because they involve a form of sequentiality [56].

7. Conclusion

Existing approaches to implementing connectors force one to make a choice between high throughput (at the cost of parallelism) and high parallelism (at the cost of throughput). In this paper, we proposed a formal basis to support a solution for this problem. We found and formalized a middle ground between those approaches by defining a new product operator on port automata (PA) and by showing that in all practically relevant cases (with respect to compiling Reo connectors), one can use this new operator instead of the existing one to get both high throughput and high parallelism in a semantics-preserving way.

Although we developed our results for PA, they generalize straightforwardly to the more powerful constraint automata [15]: the problem dealt with in this paper is essentially about synchronization, while the actual data exchanged play no role. More concretely, the premises of rules WkAgr and StAgr do not change when defining $\boxtimes$ and $\boxdot$ for constraint automata. Thus, whenever those rules are applicable to PA transitions, they are applicable also to the corresponding constraint automaton transitions. Although the conclusions of WkAgr and StAgr, in contrast, change when defining $\boxtimes$ and $\boxdot$ for constraint automata (because constraint automata have richer transition labels), those changes are exactly the same for both WkAgr and StAgr. Thus, whenever those rules are both applicable, they yield exactly the same composite transition, as in the PA case. We already worked on this topic and generalized some, but not yet all, of our results, in the context of service-oriented computing [26,57].

Our motivation for this work came from two problems that we encountered while developing Reo compilers according to the centralized implementation approach: state space explosion at compile-time and oversequentialization at run-time. The middle ground between the centralized and distributed approaches that we developed in this paper resolves those problems: restricting the composition of automata to medium automata at compile-time avoids state space explosion, while the execution of medium automata in parallel processes at run-time avoids oversequentialization. We updated our Reo-to-C compiler according to the results in this paper, and preliminary results on multicore hardware show that the generated code can outperform carefully optimized hand-crafted code [14].

While inspired by Reo, our results apply to every programming language whose programs one can describe by automata satisfying the characterizations in Section 4. One example is actor-based Rebeca systems, whose semantics one can express directly with constraint automata [16]. Another possible application of our results is *projection* in choreography languages [58–63]. A projection maps a global protocol specification among k parties, called *choreography*, to k local specifications of per-party observable behavior, called *contracts* [58,59] (or *peers* [60,61] or *end-point processes* [62,63]). 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 specifications, no projection to contracts exists that satisfies the conformance requirement. The theory presented in this paper constitutes a step in a process that may alleviate this problem by automatically inferring which communication actions need to be added to otherwise unprojectable choreographies. We make a first sketch.

Choreographies are commonly formally modeled as *labeled transition systems* (LTS) or automata. To compute a projection of a choreography involving k parties, we take such an LTS as our starting point (i.e., our approach builds on top of existing choreography models). If this LTS is finite, we translate it to a *choreography* PA (by mapping transition labels in the LTS to ports).[9] Afterward, we *decompose* the resulting "big" PA into a number of "small" PA [64]. Essentially, by recovering the internal structure of the big PA, this step reveals the previously "hidden" communication actions necessary to make the original choreography projectable. Next, we recombine the small PA into a number of *contract* PA such that for each of those PA, its input ports represent communication actions of only one party. To do this, we first apply the theory of masters, slaves, and (a)synchronous regions for computing a number of "medium" PA from the small PA using $\boxtimes$ (see Section 4.2). Subsequently, we iteratively compute $\boxtimes$ of every two medium PA whose input ports belong to the same party. Finally, we construct a number of sets of PA, each of which contains: (i) a contract PA resulting from the previous step and (ii) a number of Fifo PA such that the output port of every Fifo PA is the input port of the contract PA (see also Footnote 8). Those Fifo PA essentially represent incoming message buffers of parties. The sketched process yields l sets of PA. We conjecture that, with some extra steps skipped here for simplicity, $l = k$: we have a PA set for every party. Every such a PA set can then be compiled into the implementation of a party. Communication between PA of different sets (i.e., between different parties) has to satisfy only the local synchronization requirements imposed by $\boxdot$, which can be done relatively efficiently. The previous process is applicable also to choreographies represented as UML sequence diagrams using a translation by Meng et al. [65].

More generally, our proof method, in which we compare distributed algorithms by modeling them as different product operators on automata and studying those operators' properties, is not only effective and elegant but also—as far as we know—novel. It enables formal reasoning about distributed algorithms (in particular, reasoning about their equivalence) at a different level of abstraction than, for instance, the seminal work by Lynch [17].

[9] If the model assumes synchronous communication, we should also "desynchronize" communication actions while constructing the PA from the LTS (in a semantics-preserving way, under some equivalence).

References

[1] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366, http://dx.doi.org/10.1017/S0960129504004153.
[2] F. Arbab, Puff, the magic protocol, in: G. Agha, O. Danvy, J. Meseguer (Eds.), Talcott Festschrift, in: LNCS, vol. 7000, Springer, 2011, pp. 169–206.
[3] S. Meng, F. Arbab, Web services choreography and orchestration in Reo and constraint automata, in: A. Ricci, B. Angerer, M. Schumacher (Eds.), Proceedings of SAC 2007, ACM, 2007, pp. 346–353.
[4] S. Meng, F. Arbab, A model for Web service coordination in long-running transactions, in: X. Bai, Y. Li (Eds.), Proceedings of SOSE 2010, IEEE, 2010, pp. 121–128.
[5] S.-S. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, H. Afsarmanesh, Automatic code generation for the orchestration of web services with Reo, in: F. De Paoli, E. Pimentel, G. Zavattaro (Eds.), Proceedings of ESOCC 2012, in: LNCS, vol. 7592, Springer, 2012, pp. 1–16.
[6] S.-S. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, H. Afsarmanesh, Orchestrating web services using Reo: from circuits and behaviors to automatically generated code, Serv. Oriented Comput. Appl. 8 (4) (2014) 277–297, http://dx.doi.org/10.1007/s11761-013-0147-1.
[7] S.-S. Jongmans, F. Arbab, Modularizing and specifying protocols among threads, in: S. Gay, P. Kelly (Eds.), Proceedings of PLACES 2012, in: EPTCS, CoRR, vol. 109, 2013, pp. 34–45.
[8] S.-S. Jongmans, F. Arbab, Overview of thirty semantic formalisms for Reo, Sci. Ann. Comput. Sci. 22 (1) (2012) 201–251, http://dx.doi.org/10.7561/SACS.2012.1.201.
[9] N. Kokash, C. Krause, E. de Vink, Reo+mCRL2: a framework for model-checking dataflow in service compositions, Form. Asp. Comput. 24 (2) (2012) 187–216, http://dx.doi.org/10.1007/s00165-011-0191-6.
[10] D. Clarke, J. Proença, Partial connector colouring, in: M. Sirjani (Ed.), Proceedings of COORDINATION 2012, in: LNCS, vol. 7274, Springer, 2012, pp. 59–73.
[11] J. Proença, Synchronous coordination of distributed components, Ph.D. thesis, Leiden University, 2011, http://hdl.handle.net/1887/17624.
[12] J. Proença, D. Clarke, E. de Vink, F. Arbab, Decoupled execution of synchronous coordination models via behavioural automata, in: M.-R. Mousavi, A. Ravara (Eds.), Proceedings of FOCLASA 2011, in: EPTCS, CoRR, vol. 58, 2011, pp. 65–79.
[13] J. Proença, D. Clarke, E. de Vink, F. Arbab, Dreams: a framework for distributed synchronous coordination, in: M. Viroli, G. Castelli, J.L.F. Marquez (Eds.), Proceedings of SAC 2012, ACM, 2012, pp. 1510–1515.
[14] S.-S. Jongmans, S. Halle, F. Arbab, Reo: a dataflow inspired language for multicore, in: C. Kyriacou (Ed.), Proceedings of DFM 2013, IEEE, 2014, pp. 42–50.
[15] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in Reo by constraint automata, Sci. Comput. Program. 61 (2) (2006) 75–113, http://dx.doi.org/10.1016/j.scico.2005.10.008.
[16] M. Sirjani, M.-M. Jaghoori, C. Baier, F. Arbab, Compositional semantics of an actor-based language using constraint automata, in: P. Ciancarini, H. Wiklicky (Eds.), Proceedings of COORDINATION 2006, in: LNCS, vol. 4038, Springer, 2006, pp. 281–297.
[17] N. Lynch, Distributed Algorithms, Elsevier, 1996.
[18] S.-S. Jongmans, F. Arbab, Global consensus through local synchronization, in: C. Canal, M. Villari (Eds.), Proceedings of FOCLASA 2013, in: CCIS, vol. 393, Springer, 2013, pp. 174–188.
[19] C. Koehler, D. Clarke, Decomposing port automata, in: M. Schumacher, A. Wood (Eds.), Proceedings of SAC 2009, ACM, 2009, pp. 1369–1373.
[20] S.-S. Jongmans, C. Krause, F. Arbab, Encoding context-sensitivity in Reo into non-context-sensitive semantic models, in: W. de Meuter, G.-C. Roman (Eds.), Proceedings of COORDINATION 2011, in: LNCS, vol. 6721, Springer, 2011, pp. 31–48.
[21] M.-R. Mousavi, M. Sirjani, F. Arbab, Formal semantics and analysis of component connectors in Reo, in: C. Canal, M. Viroli (Eds.), Proceedings of FOCLASA 2005, in: ENTCS, vol. 154, Elsevier, 2006, pp. 83–99.
[22] R. Khosravi, M. Sirjani, N. Asoudeh, S. Sahebi, H. Iravanchi, Modeling and analysis of Reo connectors using alloy, in: D. Lea, G. Zavattaro (Eds.), Proceedings of COORDINATION 2008, in: LNCS, vol. 5052, Springer, 2008, pp. 169–183.
[23] T. Parr, Enforcing strict model-view separation in template engines, in: S. Feldman, M. Uretsky, M. Najork, C. Wills (Eds.), Proceedings of WWW 2004, ACM, 2004, pp. 224–233.
[24] C. Baier, V. Wolf, Stochastic reasoning about channel-based component connectors, in: P. Ciancarini, H. Wiklicky (Eds.), Proceedings of COORDINATION 2006, in: LNCS, vol. 4038, Springer, 2006, pp. 1–15.
[25] N. Kokash, B. Changizi, F. Arbab, A semantic model for service composition with coordination time delays, in: J.S. Dong, H. Zhu (Eds.), Proceedings of ICFEM, in: LNCS, vol. 6447, Springer, 2010, pp. 106–121.
[26] S.-S. Jongmans, F. Santini, F. Arbab, Partially-distributed coordination with Reo, in: M. Aldinucci, D. D'Agostino, P. Kilpatrick (Eds.), Proceedings of PDP 2014, IEEE, 2014, pp. 697–706.
[27] S.-S. Jongmans, F. Arbab, Global consensus through local synchronization, Tech. Rep. FM-1303, CWI, 2013, http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-21484.
[28] D. Clarke, D. Costa, F. Arbab, Connector colouring I: synchronisation and context dependency, Sci. Comput. Program. 66 (3) (2007) 205–225, http://dx.doi.org/10.1016/j.scico.2007.01.009.
[29] S.-S. Jongmans, D. Clarke, J. Proença, A procedure for splitting processes and its application to coordination, in: N. Kokash, A. Ravara (Eds.), Proceedings of FOCLASA 2012, in: EPTCS, CoRR, vol. 91, 2012, pp. 79–96.
[30] S.-S.T.Q. Jongmans, D. Clarke, J. Proença, A procedure for splitting data-aware processes and its application to coordination, Sci. Comput. Program. 115–116 (2016) 47–78, in this issue.
[31] B. Pourvatan, M. Sirjani, F. Arbab, M. Bonsangue, Decomposition of constraint automata, in: L. Barbosa, M. Lumpe (Eds.), Proceedings of FACS 2010, in: LNCS, vol. 6921, Springer, 2012, pp. 237–258.
[32] D. Gelernter, Generative communication in Linda, ACM Trans. Program. Lang. Syst. 7 (1) (1985) 80–112, http://dx.doi.org/10.1145/2363.2433.
[33] G. Papadopoulos, F. Arbab, Coordination models and languages, Adv. Comput. 46 (1998) 329–400, http://dx.doi.org/10.1016/S0065-2458(08)60208-9.
[34] M.D. Feng, W.F. Wong, C.K. Yuen, BaLinda lisp: design and implementation, Comput. Lang. 22 (4) (1996) 205–214, http://dx.doi.org/10.1016/S0096-0551(96)00016-1.
[35] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, T spaces, IBM Syst. J. 37 (3) (1998) 454–474, http://dx.doi.org/10.1147/sj.373.0454.
[36] M.D. Feng, Y.Q. Gao, C.K. Yuen, Distributed Linda tuplespace algorithms and implementations, in: B. Buchberger, J. Volkert (Eds.), Proceedings of CONPAR–VAPP 1994, in: LNCS, vol. 854, Springer, 1994, pp. 581–592.
[37] R. Bjornson, Linda on distributed memory multiprocessors, Ph.D. thesis, Yale University, 1993, http://cpsc.yale.edu/sites/default/files/files/tr931.pdf.
[38] A. Rowstron, A. Wood, An efficient distributed tuple space implementation for networks of workstations, in: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Proceedings of Euro-Par 1996, in: LNCS, vol. 1123, Springer, 1996, pp. 510–513.
[39] R. Menezes, R. Tolksdorf, A new approach to scalable Linda-systems based on swarms, in: H.H. Kennesaw, G. Papadopoulos (Eds.), Proceedings of SAC 2003, ACM, 2003, pp. 375–379.
[40] A. Atkinson, A dynamic, decentralised search algorithm for efficient data retrieval in a distributed tuple space, in: J. Chen, R. Ranjan (Eds.), Proceedings of AusPDC 2010, ACM, 2010, pp. 21–30.
[41] B. Bonakdarpour, M. Bozga, M. Jaber, J. Quilbeuf, J. Sifakis, A framework for automated distributed implementation of component-based models, Distrib. Comput. 25 (5) (2012) 383–409, http://dx.doi.org/10.1007/s00446-012-0168-6.

[42] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: D.V. Hung, P. Pandya (Eds.), Proceedings of SEFM 2006, IEEE, 2006, pp. 3–12.
[43] K. Dokter, S.-S. Jongmans, F. Arbab, S. Bliudze, Relating BIP and Reo, in: S. Knight, I. Lanese, A. Lluch-Lafuente, H.-T. Vieira (Eds.), Proceedings of ICE 2015, in: EPTCS, CoRR, 2015.
[44] J. Vrancken, The algebra of communicating processes with empty process, Theor. Comput. Sci. 177 (2) (1997) 287–328, http://dx.doi.org/10.1016/S0304-3975(96)00250-2.
[45] J. Bergstra, J.-W. Klop, ACP$_\tau$: a universal axiom system for process specification, in: M. Wirsing, J. Bergstra (Eds.), Algebraic Methods: Theory, Tools and Applications, in: LNCS, vol. 394, Springer, 1989, pp. 445–463.
[46] J. Baeten, R. van Glabbeek, Merge and termination in process algebra, in: K. Nori (Ed.), Proceedings of FST&TCS 1987, in: LNCS, vol. 287, Springer, 1987, pp. 153–172.
[47] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, UPPAAL — a tool suite for automatic verification of real-time systems, in: R. Alur, T. Henzinger, E. Sontag (Eds.), Hybrid Systems III, in: LNCS, vol. 1066, Springer, 1996, pp. 232–243.
[48] J. Berendsen, F. Vaandrager, Compositional abstraction in real-time model checking, in: F. Cassez, C. Jard (Eds.), Proceedings of FORMATS 2008, in: LNCS, vol. 5215, Springer, 2008, pp. 233–249.
[49] S. Anantharaman, J. Chen, G. Hains, A synchronous process calculus for service costs, in: B. Aichernig, B. Beckert (Eds.), Proceedings of SEFM 2005, IEEE, 2005, pp. 435–444.
[50] R. Segala, Modeling and verification of randomized distributed real-time systems, Ph.D. thesis, Massachusetts Institute of Technology, 1995, http://publications.csail.mit.edu/lcs/specpub.php?id=1231.
[51] B. Klin, V. Sassone, Structural operational semantics for stochastic process calculi, in: R. Amadio (Ed.), Proceedings of FOSSACS 2008, in: LNCS, vol. 4962, Springer, 2008, pp. 428–442.
[52] C. Priami, Stochastic π-calculus, Comput. J. 38 (7) (1995) 578–589, http://dx.doi.org/10.1093/comjnl/38.7.578.
[53] W.-J. Yeh, Controlling state explosion in reachability analysis, Ph.D. thesis, Purdue University, 1993.
[54] D. Kuske, I. Meinecke, Branching automata with costs — a way of reflecting parallelism in costs, in: O. Ibarra, Z. Dang (Eds.), Proceedings of CIAA 2003, in: LNCS, vol. 2759, Springer, 2003, pp. 150–162.
[55] K. Lodaya, P. Weil, Series–parallel languages and the bounded-width property, Theor. Comput. Sci. 237 (1–2) (2000) 347–380, http://dx.doi.org/10.1016/S0304-3975(00)00031-1.
[56] D. Kitchin, A. Quark, W. Cook, J. Misra, The Orc programming language, in: D. Lee, A. Lopes, A. Poetzsch-Heffter (Eds.), Proceedings of FMOODS/FORTE 2009, in: LNCS, vol. 5522, Springer, 2009, pp. 1–25.
[57] S.-S. Jongmans, F. Santini, F. Arbab, Partially-distributed coordination with Reo and constriant automata, Serv. Oriented Comput. Appl. (2015), http://dx.doi.org/10.1007/s11761-015-0177-y, in press.
[58] M. Bravetti, G. Zavattaro, Towards a unifying theory for choreography conformance and contract compliance, in: M. Lumpe, W. Vanderperren (Eds.), Proceedings of SC 2007, in: LNCS, vol. 4829, Springer, 2007, pp. 34–50.
[59] M. Bravetti, G. Zavattaro, Contract compliance and choreography conformance in the presence of message queues, in: R. Bruni, K. Wolf (Eds.), Proceedings of WS–FM 2008, in: LNCS, vol. 5387, Springer, 2009, pp. 37–45.
[60] X. Fu, T. Bultan, J. Su, Conversation protocols: a formalism for specification and verification of reactive electronic services, Theor. Comput. Sci. 328 (1–2) (2004) 19–37, http://dx.doi.org/10.1016/j.tcs.2004.07.004.
[61] X. Fu, T. Bultan, J. Su, Realizability of conversation protocols with message contents, Int. J. Web Serv. Res. 2 (4) (2005) 68–93, http://dx.doi.org/10.4018/jwsr.2005100104.
[62] M. Carbone, K. Honda, N. Yoshida, Structured communication-centered programming for web services, ACM Trans. Program. Lang. Syst. 34 (2) (2012) 8:1–8:78, http://dx.doi.org/10.1145/2220365.2220367.
[63] K. Honda, N. Yoshida, M. Carbone, Multiparty asynchronous session types, in: G. Necula, P. Wadler (Eds.), Proceedings of POPL 2008, ACM, 2008, pp. 273–284.
[64] F. Arbab, C. Baier, F.D. Boer, J. Rutten, M. Sirjani, Synthesis of Reo circuits for implementation of component-connector automata specifications, in: J.-M. Jacquet, G.P. Picco (Eds.), Proceedings of COORDINATION 2005, in: LNCS, vol. 3454, Springer, 2005, pp. 236–251.
[65] S. Meng, F. Arbab, C. Baier, Synthesis of Reo circuits from scenario-based interaction specifications, Sci. Comput. Program. 76 (8) (2011) 651–680, http://dx.doi.org/10.1016/j.scico.2010.03.002.